Numéro d'ordre : 2005-ISAL-0050 Année 2005

Thèse

Conception et mise en œuvre d'un environnement logiciel de manipulation et d'accès à des données réparties. Application aux grilles d'images médicales : Le système DSEM/DM2.

présentée devant

L'Institut National des Sciences Appliquées de Lyon

pour obtenir

Le grade de docteur

Spécialité : Informatique
École doctorale : Informatique et Information pour la Société (EDIIS)

Par

Héctor DUQUE
Master in Computer Sciences

Soutenue le 12 Juillet 2005.

Après avis de :

M. Vincent BRETON	CR, HDR
M. Kader HAMEURLAIN	Pr
M. Michel RIVEILL	Pr

Devant la Commission d'examen composé de :

Examinateur	**M. Vincent BRETON**	CR, HDR
Directeur de thèse	**M. Lionel BRUNIE**	Pr
Examinateur	**M. Kader HAMEURLAIN**	Pr
Directrice de thèse	**Mme. Isabelle MAGNIN**	DR
Examinateur	**M. Jean-Francais MEHAUT**	Pr
Examinateur	**M. Johan MONTAGNAT**	CR

CREATIS
Centre de Recherche et d'Applications en Traitement de l'Image et du Signal
LIRIS
Laboratoire d'Ingénierie des Systèmes d'Information

Résumé

Chercheurs et médecins ont besoin d'interroger de grandes collections d'images médicales par leur contenu plutôt que seulement par leurs meta-données (nom du patient, date, nom du médecin). De telles requêtes par le contenu exigent d'analyser l'image et les "objets" visibles dans l'image et éventuellement de la(les) comparer à des bases d'images de référence ou à des atlas médicaux. Ces traitements peuvent s'avérer extrêmement coûteux en terme de puissance de calcul. Dans ce cadre, les grilles proposent un paradigme architectural très prometteur en raison de leur très bon rapport performance/coût, de leur potentiel d'extensibilité, de leur richesse fonctionnelle.

Les premiers travaux sur les grilles biomédicales ont démarré seulement récemment (cf. 1ère conférence Healthgrid, Lyon, 2003). Jusqu'ici ces travaux se sont surtout concentrés sur la gridification des algorithmes de traitement de données (images, génome), sur le déploiement d'infrastructures de grilles pour la biomédecine, sur la sécurité-confidentialité des données et les problèmes éthiques. Peu de travaux se sont réellement attachés à la problématique, pourtant centrale, de l'interface entre les systèmes d'information et les bases de données médicales d'une part ; les infrastructures de grilles d'autre part.

La vision que nous défendons dans cette thèse est celle de grilles biomédicales partenaires des systèmes médicaux (hôpitaux), à la fois fournisseuses de puissance de calcul et plates-formes de partage d'informations. Notre hypothèse est que les données médicales resteront encore longtemps gérées au sein des entités des opérateurs de santé. Seules des données anonymisées (pseudomisées) dans le cadre de processus de traitement d'image, d'aide au diagnostic ou d'étude épidémiologique sont susceptibles d'être stockées sur les dispositifs de stockage de la grille.

Dans ce cadre, cette thèse propose une architecture logicielle de partage d'images médicales réparties à grande échelle. S'appuyant sur l'exis-

tence a priori d'une infrastructure de grille, nous proposons une architecture multi-couche d'entités logicielles communicantes (DSE : Distributed Systems Engines). Fondée sur une modélisation hiérarchique sémantique, cette architecture permet de concevoir et de déployer des applications réparties performantes, fortement extensibles et ouvertes, capables d'assurer l'interface entre grille, systèmes de stockage de données et plates-formes logicielles locales (propres aux entités de santé) et dispositifs d'acquisition d'images, tout en garantissant à chaque entité une maîtrise complète de ses données dont elle reste propriétaire.

Sur un plan conceptuel, DSE s'appuie sur une décomposition sémantique et opérationnelle des applications. Cette structuration verticale (selon le niveau de complexité sémantique) et horizontale (selon le type de service fourni) définit un modèle de conception et de mise en œuvre à la fois extensible (ajout d'"outils" ou de "services"), interopérable (définitions de "drivers" transactionels ou de service), ouvert (appels à des services externes). Elle permet également d'intégrer les grilles comme des partenaires naturels de l'application, au même titre, par exemple, que les serveurs locaux d'images médicales.

S'appuyant sur ce modèle architectural, nous avons conçu et implémenté une plate-forme logicielle (DSEM-DM^2) dédiée au partage d'images médicales à large échelle. Cette plate-forme permet l'interrogation et la recherche de grandes bases de données d'images via des requêtes hybrides (c'est-à-dire intégrant traitements sur l'image et requête sur ses méta-données). Elle a été conçue pour permettre le déploiement des traitements d'images sur une grille partenaire.

Des expérimentations ont été menées pour évaluer l'efficacité et la faisabilité de l'approche proposée dans DSEM-DM^2. Les premiers résultats obtenus sont très encourageants.

Abstract

Due to increasing medical requirements, researchers and physicians need to query large medical image data sets by their content, rather than only by their associated metadata. These queries involve a vast quantity of data analysis and require high computing power. Therefore using grid technology arises as promising solution not only for higher performance but also for overcoming scalability and extensibility issues.

Currently, ongoing works deal with either grid middleware development, static and centralized medical image databases processing, specialized algorithms for querying images by their content, security and confidentiality, or ethical issues. However, they partially cover the main problem of allowing the Information Systems and Databases to interface within a high distributed environment constituting the grid.

Our vision, in this thesis, is the one of a bio-medical grid as a partner of hospital's information systems, sharing computing resources as well as a platform for sharing information. Therefore, we aim at (i) providing transparent access to huge distributed medical data sets, (ii) querying these data by their content, and (iii) sharing computing resources within the grid. Our main hypothesis is to keep the medical data within their entities, use and store only anonymous data for image processing, medical diagnosis, or epidemiological studies.

*Assuming the existence of a grid infrastructure, we suggest a multi-layered architecture (**Distributed Systems Engines - DSE**). This architecture allows us to design High Performance Distributed Systems which are highly extensible, scalable and open. It ensures the connection between the grid, data storing systems, and medical platforms.*

The conceptual design of the architecture assumes a horizontal definition for each one of the layers, and is based on a multi-process structure. This structure enables the exchange of messages between processes by using the Message Passing Paradigm. These processes and messages

allow one to define entities of a higher level of semantic significance, which we call ***Drivers*** *and, which instead of single messages, deal with different kinds of* ***transactions : queries, tasks and requests****. Thus, we define different kinds of drivers for dealing with each kind of transaction, and in a higher level, we define* ***services*** *as an aggregation of drivers. This architectural framework of drivers and services eases the design of components of a Distributed System (DS), which we call* ***engines****, and also eases the extensibility and scalability of the DS.*

The ***DSE architecture*** *groups its layers in two big types : middleware and application layers. The middleware layers (the first three ones) have been implemented in a prototype (****DSEM****), and then used for developing a medical application (****Distributed Medical Data Manager (DM^2)****. Our architectural framework (****DSE****) and its implemented prototype (****DSEM, DM^2****) have been tested in a stressing environment. The medical problem consists on offering a* ***Grid Service*** *for solving queries by-content and hybrid queries over huge datasets of medical images. The issues of concurrency, transparency and location independence have all been addressed, and the experiment results are promising.*

Acknowledgments

This work was partly supported by the European DataGrid IST project, the French ministry ACI-GRID project, and the ECOS Nord Committee (action C03S02).

The CNRS Rhone-Alpes, INSA of Lyon, CREATIS and LIRIS laboratories have provided the physical and human resources in order to finish with success this thesis.

–

"El hombre es esclavo de sus palabras y dueño de sus silencios.", ***anónimo***

–

Table des matières

–

Table des figures

Chapitre 1

Résumé Étendu.

Ce résumé est destiné aux lecteurs francophones. Il vise à donner une idée globale du contenu de cette thèse. Nous prions le lecteur interessé par plus de détails de se reporter au manuscrit en anglais.

1.1 Introduction

Chercheurs et médecins ont besoin d'interroger de grandes collections d'images médicales par leur contenu image (présence d'objets d'intérêt, textures, gradients d'opacité, mesures de données physiques (ex : volume)) et leurs meta-données (nom du patient, date, nom du médecin...). De telles requêtes dites "requêtes hybrides" (ou "requêtes par le contenu") exigent d'analyser l'image et les "objets" visibles dans l'image et éventuellement de la(les) comparer à des bases d'images de référence ou à des atlas médicaux.

Ces traitements peuvent s'avérer extrêmement coûteux en terme de puissance de calcul. Dans ce contexte, les grilles apparaissent aujourd'hui comme un paradigme architectural très prometteur en raison de leur très bon rapport performance/coût, de leur potentiel d'extensibilité et de leur richesse fonctionnelle.

Les premiers travaux sur les grilles biomédicales ont démarré seulement récemment (cf. 1ère conférence Healthgrid, Lyon, 2003). Jusqu'ici ces travaux se sont surtout concentrés sur la "gridification" des algorithmes de traitement de données (images médicales, génome...), sur le déploiement d'infrastructures de grilles pour la biologie et la médecine, sur la sécurité (confidentialité) des données et les problèmes éthiques... Peu de travaux se sont concentrés sur la problématique, pourtant centrale, de l'interface entre d'une part, les systèmes d'information et les bases de données médicales utilisés par les hôpitaux ; d'autre part, les infrastructures de grilles.

Cette thèse se propose d'étudier cette problématique.

La vision que nous défendons est celle de grilles biomédicales "partenaires" informatiques des systèmes médicaux (hôpitaux), à la fois fournisseuses de puissance de calcul et plates-formes de partage d'informations. Notre hypothèse est que les données médicales resteront encore longtemps gérées au sein des systèmes informatiques des opérateurs de santé. Seules des données anonymisées (pseudonimisées), ou des données cryptées, dans le cadre de processus de traitement d'image, d'aide

au diagnostic ou d'études épidémiologiques, seront susceptibles d'être copiées sur les dispositifs de stockage de la grille.

Dans ce cadre, cette thèse propose une architecture logicielle de partage d'images médicales réparties à grande échelle. S'appuyant sur l'existence a priori d'une infrastructure de grille, nous proposons une architecture multi-couche fondée sur la définition et la mise en place d'entités logicielles communicantes (DSE : Distributed Systems Engines). Proposant une modélisation hiérarchique sémantique, cette architecture permet de concevoir et de déployer des applications réparties performantes, fortement extensibles et ouvertes, capables d'assurer l'interface entre grille, systèmes de stockage de données, plates-formes informatiques hospitalières et dispositifs d'acquisition d'images, tout en garantissant à chaque acteur une maîtrise complète de ses données dont il reste le seul propriétaire.

Sur un plan conceptuel, l'architecture DSE s'appuie sur une décomposition sémantique et opérationnelle des applications. Cette décomposition verticale (selon le niveau de complexité sémantique) et horizontale (selon le type de service fourni) définit un modèle de conception et de mise en œuvre à la fois extensible (ajout d'"outils" ou de "services" ; définition de "drivers" transactionels) et ouvert (appel à des services externes). Elle permet également d'intégrer les grilles comme des partenaires naturels de l'application, au même titre, par exemple, que les serveurs locaux d'images médicales.

S'appuyant sur ce modèle architectural, nous avons conçu et implémenté une plate-forme logicielle (DSEM-DM2) dédiée au partage d'images médicales à large échelle. Cette plate-forme offre des fonctionnalités d'interrogation de grandes bases de données d'images via des requêtes hybrides. Sur un plan opérationnel, elle a été conçue pour permettre le déploiement des traitements d'images associés aux requêtes sur une grille partenaire.

Des expérimentations ont été menées pour évaluer l'efficacité et la faisabilité de l'approche proposée dans DSEM-DM2. Les premiers résultats obtenus sont tout à fait encourageants.

Ce résumé étendu, qui reprend la structure du manuscrit rédigé en anglais comporte 6 chapitres outre cette introduction. La section 1.2 décrit les cas d'utilisation cibles de cette thèse, sur lesquels nous nous appuierons pour valider l'applicabilité des concepts et outils proposés. La section 1.3 présente l'état de l'art des technologies et domaines de recherche connexes à nos travaux : intergiciels[1] de grille, technologies d'intégration de services répartis, dispositifs de stockage d'images. La section 1.4 présente l'architecture DSE. La conception et l'implémentation de la plate-forme DSEM-DM2 sont étudiées du section 1.5. Les tests de performance que nous avons menés ainsi qu'un scénario d'utilisation de la plate-forme DSEM-DM2 sont présentés et analysés section 1.6. Une discussion de nos propositions en regard des cas d'utilisation cibles et de l'état de l'art ainsi que les principales perspectives à ces travaux concluent ce document de synthèse.

[1] *Intergiciel* est la traduction la plus couramment admise du terme anglais *Middleware*.

1.2 Contexte applicatif

De nombreuses applications médicales utilisent des requêtes par le contenu : aide au diagnostic, suivi des patients, épidémiologie, formation médicale... Pour illustrer les problématiques mises en œuvre, nous considérerons le cas d'utilisation suivant.

Un cardiologue, dans le cadre du diagnostic d'un patient, recherche des images cardiaques d'autres patients du même âge, pour lesquels un diagnostic a été confirmé, similaires à celles de son patient, c'est-à-dire, de manière plus précise, des images présentant une "fraction d'éjection"[2] supérieure à 0,5. Les images sélectionnées seront classées en fonction de leur degré de similarité. Le cardiologue pourra alors choisir les images qui l'intéressent le plus, les visualiser ainsi que le dossier médical des patients concernés (diagnostic, protocole thérapeutique mis en œuvre...). Ces dossiers seront anonymisés si le cardiologue n'est pas le médecin traitant des patients concernés. Enfin, le cardiologue rédigera son diagnostic qui sera intégré au dossier du patient et stocké dans la base de données avec l'examen (images) du patient.

Fonctionnellement, exécuter cette "transaction" nécessite :

- d'exécuter des requêtes sur des bases réparties de metadonnées médicales (âge du patient, fraction d'éjection lorsqu'elle est connue) ;
- de pouvoir accéder aux systèmes de stockage d'images et de dossiers médicaux distribués sur différents hôpitaux ;
- d'exécuter des traitements d'images complexes sur les images potentiellement pertinentes (calcul de la fraction d'éjection lorsqu'elle n'est pas connue) en quasi temps réel (temps de réponse de quelques secondes). Ceci impose d'être capable de mobiliser une grande puissance de calcul. Notre proposition est, pour ce faire, d'utiliser les ressources d'une grille partenaire ;
- d'anonymiser des données confidentielles ;
- de mettre à jour le dossier médical du patient et d'archiver les images du patient dans un système ad hoc.

L'objectif de cette thèse est, sur un plan théorique, de proposer un cadre méthodologique et architectural à même de gérer des applications présentant de telles exigences fonctionnelles ; sur un plan applicatif, de développer un prototype, fondé sur cette approche, afin d'évaluer la faisabilité de nos propositions.

Cette thèse d'intègre dans les projets ACI Grid Medigrid (étude de l'utilisation de grilles de calcul pour le traitement d'images médicales), Ragtime (projet rhône-alpin visant à déployer une grille biomédicale à l'échelle de la Région), Datagrid et EGEE (projets européens IST de grille à l'échelle européenne.)

1.3 Etat de l'art

Nos travaux se situent à l'intersection des grilles de calcul, du développement et de l'intégration de services et composants répartis, du stockage de données et du traitement d'images médicales. Nous proposons donc dans le manuscrit une revue des principaux travaux et projets dans ces domaines.

[2]La fraction d'éjection désigne la fraction du flux sanguin pompée par le ventricule gauche à chaque pulsation.

Les intergiciels de grille peuvent être classés en trois grandes catégories en fonction de leurs cibles applicatives, de l'infrastructure de calcul et de celle de partage de données sous-jacente [4] :

1. Les intergiciels de grille orientés calcul (*Computational grids*) : ces intergiciels (Globus dans ses premières versions [14], Legion [128] [62], Unicore [165] [65], etc.) visent principalement à permettre l'exécution de calculs parallèles complexes sur des dispositifs de traitement intensif (clusters, supercalculateurs...) répartis à grande échelle. Les données manipulées dans ces calculs sont stockées sur la grille. L'attention est notamment placée sur l'ordonnancement de tâches et la réplication des données. Les utilitaires de manipulation de données ainsi que les mécanismes de gestion de la confidentialité et de partage d'information (metadonnées) sont généralement assez basiques.
2. Les intergiciels de calcul global (*Scavenging grids*) : ces intergiciels (Condor [127], BOINC [202], etc.) permettent d'exécuter des codes faiblement couplés, voire non couplés sur un ensemble potentellement très important d'ordinateurs personnels ou de stations de travail (on parle aussi d'"Internet computing") dont on utilise les cycles de calcul laissés disponibles par les applications s'exécutant sur ces machines.
3. Les intergiciels de grille orientés données (*Data grids*) : ces intergiciels (Globus [129], SRB [166], DataGrid [126], etc) mettent l'accent sur la gestion de données partagées au sein de communautés virtuelles. Ils proposent des fonctionalités de gestion de grands volumes de données réparties à grande échelle (indexation, réplication, sécurité, cache...).

Notre objectif n'est pas de proposer un nouvel intergiciel de grille ni même d'apporter de nouvelles fonctionnalités à un intergiciel existant. Nous faisons l'hypothèse (réaliste au moins à moyen terme) de l'existence d'infrastructures (matérielles et logicielles) de grille accessibles aux acteurs d'un réseau de santé ou d'une communauté d'utilisateurs médicaux. Sous cette hypothèse, notre objectif est de proposer une architecture et les outils logiciels nécessaires à la connexion des systèmes informatiques de ces utilisateurs à la grille. Par connexion, nous entendons la possibilité d'utiliser les services proposés par la grille : exécution de calculs, partage d'information, authentification, réplication... Il est donc important de bien analyser les intergiciels de grille, leurs fonctionnalités et leur structure, afin d'être capable d'interagir et de s'interfacer de manière efficace avec ces intergiciels.

L'intergiciel de grille le plus utilisé aujourdhui est sans conteste le Globus Toolkit [129] [14]. Initié en 1996, Globus vise dans ses versions 3 et 4 le déploiement de grilles à grande échelle multi-institutionnelles. Globus est conçu autour d'une architecture en couches intégrant des services (conformes à WSRF dans la version 4 de Globus) hiérarchisés en fonction de leur portée (de la ressource de calcul à la communauté virtuelle). Globus offre une pallette d'outils et de kits de développement très large.

L'évolution des différentes versions de Globus illustre bien le processus de normalisation vers lequel convergent les intergiciels de grille. En effet, après une époque de "bouillonnement" durant laquelle chaque intergiciel était conçu comme un logiciel "propriétaire", le GGF (Global Grid Forum) a proposé de mettre en place un standard de développement de composants de grille. OGSA (Open Grid Services

Architecture)[162] [58] s'appuie ainsi sur la normalisation d'une infrastructure de services de grille de base (OGSI : Open Grid Services Infrastructure, chargée en particulier de la création, du nommage, de la destruction des services) et sur la spécification de normes et conventions d'interopérabilité entre services permettant à un programmeur d'interfacer ses développements avec un intergiciel existant.

Encore plus récemment, WSRF (Web Service Resource Framework) [59] propose une infrastructure proche de l'architecture de services d'OGSI mais en se fondant sur les normes et constructions logicielles développées dans le cadre des Web services, dans une convergence des technologies de service de grille et de service Internet.

Outre Globus, on peut également citer parmi les principaux intergiciels de grille Condor [127], environnement utilisé depuis la fin des années 80 pour des applications de calcul global. Condor permet de gérer de très grands volumes de données dans le cadre d'applications impliquant des codes faiblement couplés appliqués à des flux de données importants. Un portage de Condor au-dessus de Globus (Condor-G) a récemment été réalisé [62]. Legion[63] propose un système d'exploitation de grille orienté-objet. Processeurs, périphériques de stockage, ressources matérielles ou logicielles sont modélisés comme des objets à partir desquels le programmeur va construire son application. Legion fournit des mécanismes d'adressage et de gestion de ces objets dans une vision globale unique de l'ensemble des ressources présentes sur la grille. Citons enfin, parmi les intergiciels orienté calcul, Unicore [65] qui, s'appuyant sur une architecture 3-tiers, propose un environnement sécurisé d'exécution de codes répartis offrant notamment des fonctionnalités de migration de tâches. D'autres intergiciels per se (Gridbus, SRB) sont analysés dans le corps du manuscrit.

Orienté vers l'intégration de données réparties, SRB (Storage Resource Broker)[166] est un intergiciel client-serveur fournissant un accès uniforme à des ressources et périphériques de stockage variés dans le cadre d'environnements de calcul hétérogènes. SRB supporte des systèmes de stockage comme HPSS (cf. ci-dessous) ou des systèmes de gestion de bases de données. SRB organise les données sous la forme de collections hiérarchiques et implémente des mécanismes d'association nom logique-adresse physique des données.

S'appuyant sur ces intergiciels de grilles, nous décrivons dans le corps du manuscrit plusieurs projets d'infrastructures de grille représentatifs des axes de développement actuels (Datagrid, Egee, Eurogrid, IPG, K*Grid, Crossgrid, DCGrid). Nous analysons également les principaux projets d'environnements haute performance pour l'imagerie médicale (Mammogrid, e-Diamond, Dismedi, Camaec, medGift, CasImage, Ptm3D, Aquatics, Irma, Assert). Nous renvoyons le lecteur au chapitre **Etat de l'art** du manuscrit pour l'étude de ces projets.

Situés à l'interface entre les grilles et les systèmes répartis, nos travaux s'inscrivent naturellement dans le champ des environnements d'intégration de services et de calcul réparti. Plusieurs technologies sont aujourd'hui disponibles. Citons notamment CORBA, DCOM, Java/RMI, MDA/OMA [191] . Le composant central de CORBA[134] [104] est l'ORB (Object Resource Broker). L'ORB fournit l'infrastructure de communication entre les objets manipulés : identification, localisation, transfert de données. CORBA normalise également un langage de description d'interfaces qui permet de spécifier les API des méthodes mises à disposition par les applications et services. DCOM[192] [100] offre des mécanismes de communication inter-

domaine à des composants logiciels Microsoft COM. Java/RMI (Remote Method Invocation)[193] permet d'invoquer des méthodes s'exécutant sur des machines virtuelles Java distantes via des mécanismes de sérialisation d'objet. Enfin, MDA/OMA (Model Driven Architecture/Object Management Architecture)[191] s'appuie sur un ORB CORBA pour modéliser et déployer des applications multi-composants.

Ces environnements et intergiciels offrent des outils puissants d'interopérabilité au niveau programmation. Par contre, vis-à-vis de nos applications-cibles, ils se situent à un niveau opérationnel très bas et n'offrent pas de formalisme architectural adapte à la structure effective de ces applications. Très génériques et découplés de notre contexte applicatif, ils ne permettent pas d'appréhender l'organisation logique et fonctionnelle des applications médicales cibles de nos travaux. Ils offrent par contre des outils utilisables pour l'implémentation des mécanismes de communication et d'interopérabilité inter-processus.

Enfin, nous proposons dans le corps du manuscrit une revue des systèmes de stockage réparti : systèmes de fichiers répartis, systèmes de stockage hiérarchiques (Castor, Endstore, Eurostore, HPSS), systèmes d'archivage d'images médicales (PACS) ; nous décrivons également (de maniére suscinte) le standard de description et d'échange d'images médicales DICOM3. En effect bien que nos travaux se situent dans des champs de recherche différents, nous sommes cependant appelés à utiliser des systèmes de stockage de données et à manipuler des images médicales. Il nous est donc apparu nécessaire de donner au lecteur du manuscrit les élements fondamentaux de ces technologies, indispensables pour situer le contexte applicatif de nos travaux.

1.4 L'architecture DSE (Distributed System Engines)

L'objectif de nos travaux est de fournir un cadre architectural modélisant des applications complexes (cf. cas d'utilisation décrit plus haut) faisant intervenir d'une part une grille, agissant comme support de calcul, de partage d'information et de mécanismes d'authentification ; d'autre part des sites (ex : hôpitaux) membres d'une communauté virtuelle gérant des données multimédias complexes sensibles. Dans le cadre de notre application-cible, ces données sont issues d'imageurs (tomodensitomètres X (scanners), IRM, etc.) ou de dossiers médicaux. Elles sont gérées dans des systèmes d'archivage d'images médicales (PACS) et des SGBD et soumises à des contraintes de confidentialité très strictes. En ce sens, il n'est pas envisageable de les copier (sans anonymisation) sur la grille en vue d'un partage au sein de la communauté des utilisateurs. Les traitements-cibles sur ces données sont des recherches par le contenu dans les bases d'images.

Notre hypothèse de travail est de laisser les données dans les systèmes internes aux sites partenaires. Cette hypothèse se justifie par la nécessité de maintenir la confidentialité des données, par la difficulté conséquente d'obtenir les autorisations administratives nécessaires, par l'exigence d'une cohérence très forte, permanente, des données (difficile à mettre en place dans le cadre d'une grille), par la forte localisation des accès (la plupart des demandes de lecture des données d'un patient proviennent de l'hôpital où il est soigné).

Ceci impose de mettre en place au niveau de chaque site partenaire (désigné dans

la suite sous le terme générique d'*hôpital*) une infrastructure logicielle "passerelle" (appelée dans la suite DSE : distributed system engine) entre le réseau local et la grille. Cette infrastructure passerelle doit permettre les flux de requêtes et de données de la grille et des autres hôpitaux vers l'hôpital local et réciproquement de l'hôpital vers la grille et les autres hôpitaux. Elle doit également permettre l'intégration de services variés : traitement d'images et indexation, bases de données, caches de requêtes et de données, contrôle d'accès, anonymisation, PACS...

Nous proposons de modéliser et de structurer un DSE en composants fonctionnels hiérarchiques (figure 1.1). Chaque couche hiérarchique correspond à un niveau sémantique, des processus jusqu'aux interfaces utilisateur.

Le niveau DSE^0 (échange de messages) désigne le niveau des processus. S'exécutant localement, ces processus communiquent entre eux via des mécanismes d'échange de messages fournis par un noyau de communication. Le niveau DSE^1 (transaction) correspond au traitement de requêtes complexes. Le traitement d'une requête hybride, par exemple, exige l'exécution d'un nombre important de processus (contrôle d'accès, cache, connexion à la base de metadonnées, connexion au PACS, traitement d'images, monitoring et traçabilité, etc.), qu'il faut donc savoir décrire et coordonner : c'est le rôle de ce niveau. Le niveau DSE^2 concerne l'exécution de requêtes réparties sur plusieurs DSEs. Enfin, les niveaux DSE^3 et DSE^4 représentent les interfaces applicatives (API) et utilisateur (portail, interface graphique...).

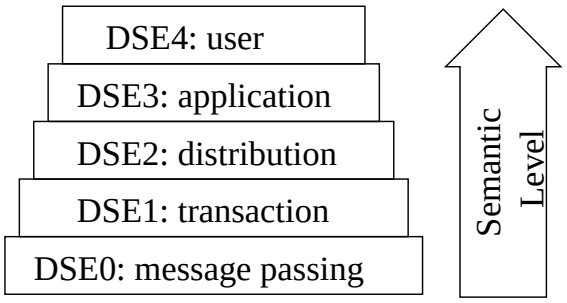


FIG. 1.1 – L'architecture multi-couche DSE

DSE^0 : échange de messages

A la base de tout applicatif réparti se trouvent des processus communicants. Le cœur de DSE^0 est donc constitué d'un noyau logiciel d'échanges de messages (MPK : Message Passing Kernel). Ce noyau est chargé de fournir des mécanismes efficaces d'échange de données entre les processus s'exécutant dans le DSE. Rappelons qu'un *DSE* peut être constitué d'un ensemble de serveurs et d'applications répartis sur une entité locale d'administration (typiquement, le réseau local d'un hôpital). Le MPK est notamment chargé du routage et de la gestion des communications ainsi que de la communication avec l'extérieur (grille, réseau métropolitain, Internet...). Différentes approches sont envisageables, notamment l'utilisation de librairies d'échanges de messages (type PVM ou MPI) et l'utilisation de services de communication inter-processus (IPC) internes au système d'exploitation. Neanmoins, l'utilisation de librairies telles que PVM (orientée multi-plateforme) ou MPI

(offrant des fonctionnalités d'échange de messages avancées), d'un coût d'exécution élévé, ne s'impose d'évidence pas lorsque les processus sont localisés sur une même machine ou sur un réseau homogène, en raison du surcoût qu'elles entraînent.

DSE[1] : transactions

Les processus du niveau 0 peuvent être assemblés pour constituer des *transactions*. De manière classique, une transaction est constituée de sous-opérations. Une transaction doit verifier les propriétés ACID : Atomicité, Consistance, Isolation et Durabilité[1]. En terme d'applicatif-cible, une transaction correspond par exemple à l'exécution d'une requête hybride locale dont on a vu plus haut qu'elle pouvait impliquer de nombreux processus.

Nous proposons de formaliser trois types de transactions (figure 1.2) : les *Queries*, les *Tasks* et les *Requests*[3]. Conceptuellement, une *Query* est constituée d'un ensemble de *Tasks et de Requests* pouvant être exécutées en parallèle ou séquentiellement. Les *Tasks* sont constituées d'un ensemble de *Requests* s'exécutant en parallèle ou en séquence . Enfin, les *Requests* exécutent des actions simples via l'échange de messages avec des serveurs internes à l'hôpital ou distants, éventuellement accesibles via la grille.

De manière plus illustrative, on utilisera par exemple une *Request* pour modéliser un accès à un service de bases de données. La *Request* sera ainsi chargée de gérer la connexion avec le SGBD, de transmettre la requête dans un format adapté (on n'utilisera pas la même *Request* selon la base de données considérée), de récupérer les résultats et éventuellement de les transformer dans un format spécifique.

Les *Tasks* offrent des possibilités de parallélisme (figures 1.2i et 1.2ii) et de répartition. Une *Task* peut être modélisée, par exemple, pour décrire le mécanisme d'accès à un fichier image. En entrée, la *Task* ne dispose que d'un identifiant logique. Son rôle est alors, à l'aide de cet identifiant, de récupérer une copie de l'image aussi efficacement que possible. La *Task* peut être ainsi amenée à vérifier la présence de l'image dans un cache, à récupérer l'image dans un PACS local, ou à contacter le service d'information de la grille pour localiser le fichier image puis soumettre un *job* de transfert de fichier à la grille. Enfin, au niveau sémantique le plus élévé, les *Queries* modélisent les transactions applicatives les plus complexes impliquant des protocoles avancés, telles que l'exécution d'une requête par le contenu.

Nous proposons d'introduire un dernier type de transaction, les *Tools*. Les applicatifs-cibles sont amenés à utiliser des services génériques, indépendants de l'applicatif, chargés de fonctions globales au DSE ou transversales à plusieurs applications. Citons par exemple les services de cache de données, de monitoring, de contrôle d'accès, de traitement d'images... Ces *Tools* indépendants de l'applicatifs peuvent être accédés par les *Queries*, les *Tasks* et les *Requests*.

Afin de distinguer les processus de niveau 0 des processus implémentant des transactions, nous appelerons ces derniers des *Drivers* (on parlera ainsi de *Query driver* (QUD), de *Task driver* (TKD), de *Request driver* (RQD) et de *Tool driver* (TOD).

[3]Nous maintenons la terminologie anglophone afin de garder une cohérence entre ce document de synthèse et le manuscrit en anglais.

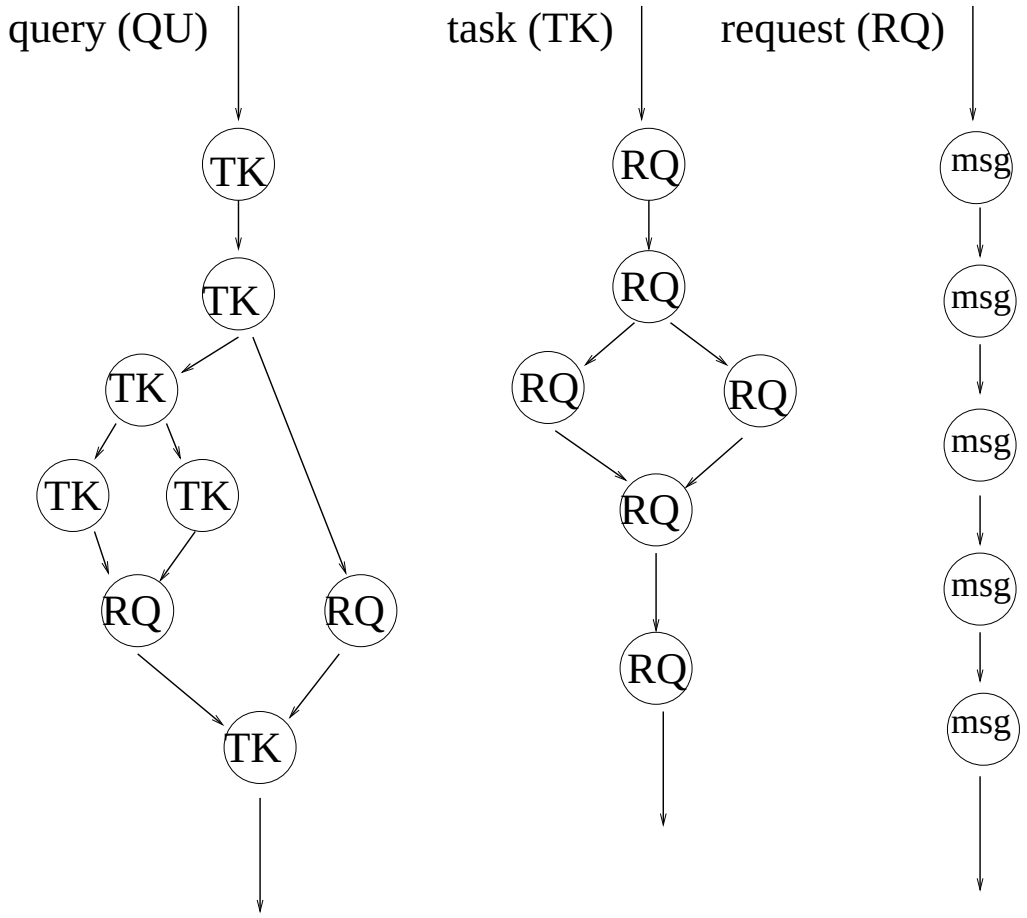


FIG. 1.2 – Types de transactions.
(i) Query, (ii) Task, (iii) Requests.

Ainsi, de manière opérationnelle, lorsqu'un DSE reçoit un message porteur d'une requête hybride : (i) Le message est transféré au *Query driver* concerné qui lance aussitôt une requête (query), (ii) La requête démarre différentes tâches concurrentes (ex : accès à un fichier image local et au dossier patient associé) via des *Task drivers* différents, (iii) Pour s'éxécuter, chaque tâche fait appel aux *Requests drivers* pertinents (ex : accès aux images) (iv) Les *Request drivers* ouvrent des connexions et échangent des messages avec les services concernés (ex : PACS), (v) Les *drivers*, dans leur exécution et quel que soit leur type, font appel aux *Tools* fournis par la plate-forme DSE (ex : cache de données). Nous renvoyons le lecteur au section 1.2 pour un exemple plus détaillé.

DSE^2 : Distribution

Le niveau DSE^2 (Distribution) apporte des fonctionnalités d'exécution répartie : localisation des données et des services, transferts de requêtes, collaboration entre DSEs... Ce niveau a naturellement vocation à s'interfacer avec les services de la grille. C'est en effet le rôle des intergiciels de grille de fournir ce type de mécanismes et de services. Dans des contextes applicatifs différents sans infrastructure de grille,

ce niveau de distribution peut être implémenté via des mécanismes de type pair-à-pair décentralisé ou le déploiement de services d'annuaires (LDAP par exemple) ad hoc.

De nouveaux composants sont définis à ce niveau, tels que les *SDA (Service Daemons)* et les *SDR (Service Drivers)*. Ces composants, construits au-dessus de ceux du niveau DSE[1], permettent de gérer l'accès aux services distribués offerts par l'application, leur coordination et leur interoperabilité.

Niveaux supérieurs

Le niveau DSE[3] (Application) définit un DSE comme un ensemble de services accessibles via des interfaces de programmation (API). Un service de ce niveau est une entité fonctionnelle indépendante utilisable par des applications. Une application est définie ainsi comme un ensemble de services coordonnés répartis sur des DSEs disséminés sur la grille (ou le réseau).

Enfin le niveau DSE[4] (Utilisateur) définit les modes d'interaction de l'utilisateur avec l'application (saisie des informations (ex : interfaces graphiques, portail), procédure d'authentification, connexion au système...)

Synthèse

L'architecture *DSE* propose un formalisme de modélisation d'applications réparties complexes. Ce formalisme suggère de structurer les applications selon une architecture fonctionnelle hiérarchique. Par sa versatilité et son adaptabilité, DSE est particulièrement bien adapté à des grilles biomédicales qui impliquent des serveurs de données et de traitement très divers, sensibles en terme de sécurité, relativement cloisonnés (peu intégrés) et situés au-delà de la frontière de la grille. Par sa démarche analytique, DSE incite les programmeurs à définir des composants génériques (noyau au niveau 0, drivers et tools au niveau 1, services aux niveaux 2 et 3) réutilisables et facilement adaptables pour s'intégrer dans plusieurs applications.

1.5 DM^2 : Distributed Medical Data Manager

Nous avons développé un prototype fondé sur DSE pour valider nos propositions et vérifier la faisabilité (en terme de performance et de fonctionnalités) d'un système de requêtes par le contenu réparties fondé sur l'utilisation d'un intergiciel de grille. La figure 1.3 illustre le positionnement de DM^2 à l'interface des serveurs de données hospitaliers et de la grille.

1.5.1 DM^2 : composants logiciels

DM^2 est composé (figures 1.4 et 1.5) d'un ensemble de *request drivers (RQD)* chargés de la connexion aux services de la grille ; d'un DICOM *request driver* chargé de l'interface avec le serveur d'images médicales DICOM de l'hôpital ; d'un *request driver* chargé de la connexion au SGBD stockant les métadonnées décrivant les images.

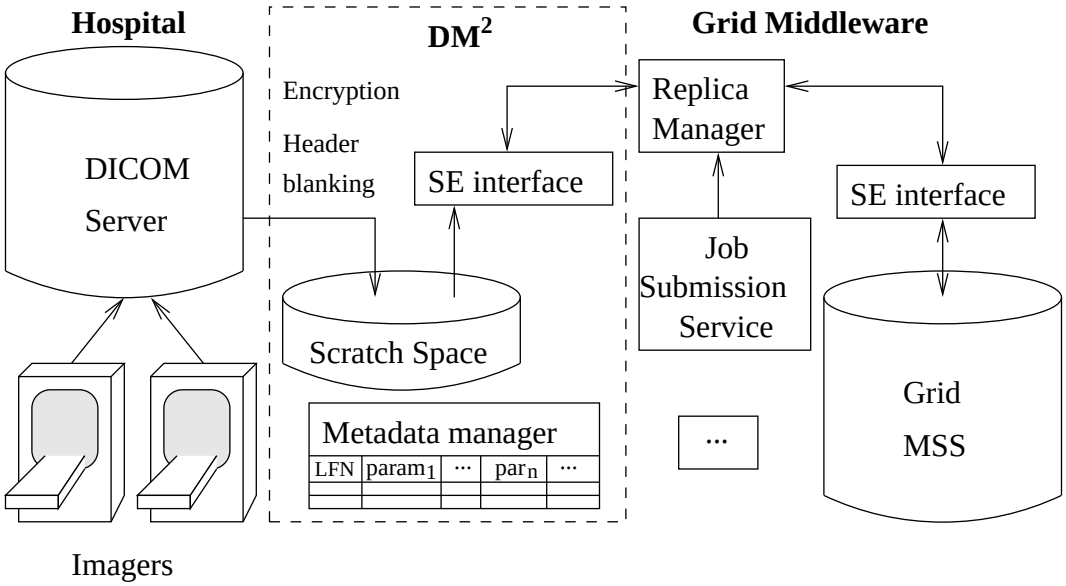


FIG. 1.3 – Architecture du système DM^2.
SE : storage element, *LFN* : logical file name

Chacun de ces *request drivers* est associé à un *task driver (TKD)* pour permettre une parallélisation des demandes. Le DICOM *task driver* est ainsi capable de transférer plusieurs images DICOM simultanément.

Au niveau DSE^2, un démon de communication reçoit les messages émanant de la grille et lance l'exécution des requêtes. Ce démon est également chargé de la gestion de la charge du serveur d'images (gestion de files d'attente si le serveur atteint sa capacité-crête). DM^2 stocke provisionement les images dans le "scratch space" avant de les transférer (éventuellement aprés anonymisation) vers la grille.

1.5.2 Mise en œuvre

Reprenons le cas d'utilisation décrit en section 1.2.

Les figures 1.4 et 1.5 illustrent la mise en œuvre de la requête du cardiologue.

Tout d'abord, le cardiologue envoie une requête (pour trouver l'IRM de Monsieur X acquise la veille dans l'hôpital) à l'aide de l'interface utilisateur DM^2. Le *DM^2 Query Driver (QUD)* envoie la requête à l'interface de la grille pour aller interroger le grid service de métadonnées. Cette requête d'interrogation est assemblée et envoyée sur la grille grâce aux *Metadata Request Driver (RQD)* et *Metadata Task Driver (TKD)*. Les autorisations d'accès de l'utilisateur aux données sont vérifiées par le système de sécurité (*Security Tool Driver (TOD)*) (étape 1 de la figure 1.4a)) ; l'identifiant logique du fichier du patient ainsi que les métadonnées qui lui sont associés (modalité d'imagerie, région d'intérêt, séquence dynamique, paramètres d'acquisition IRM, etc.) sont, une fois retournés par le service de métadonnées de la grille, renvoyés à l'interface utilisateur via le DM^2 QUD.

Une requête est ensuite effectuée pour trouver toutes les images (étape 3 de la figure 1.4b) similaires à l'image IRM d'intérêt (même région du corps, même

modalité d'acquisition...) et pour lesquelles le diagnostic médical est connu. La couche 2 de DM^2 est utilisée dans ce cas pour distribuer les requêtes sur tous les hôpitaux équipés de services de métadonnées à l'aide du DM^2 RQD et du Metadata TKD. Les identifiants logiques de toutes les images correspondant aux paramètres du fichier source du patient sont alors renvoyés aux utilisateurs

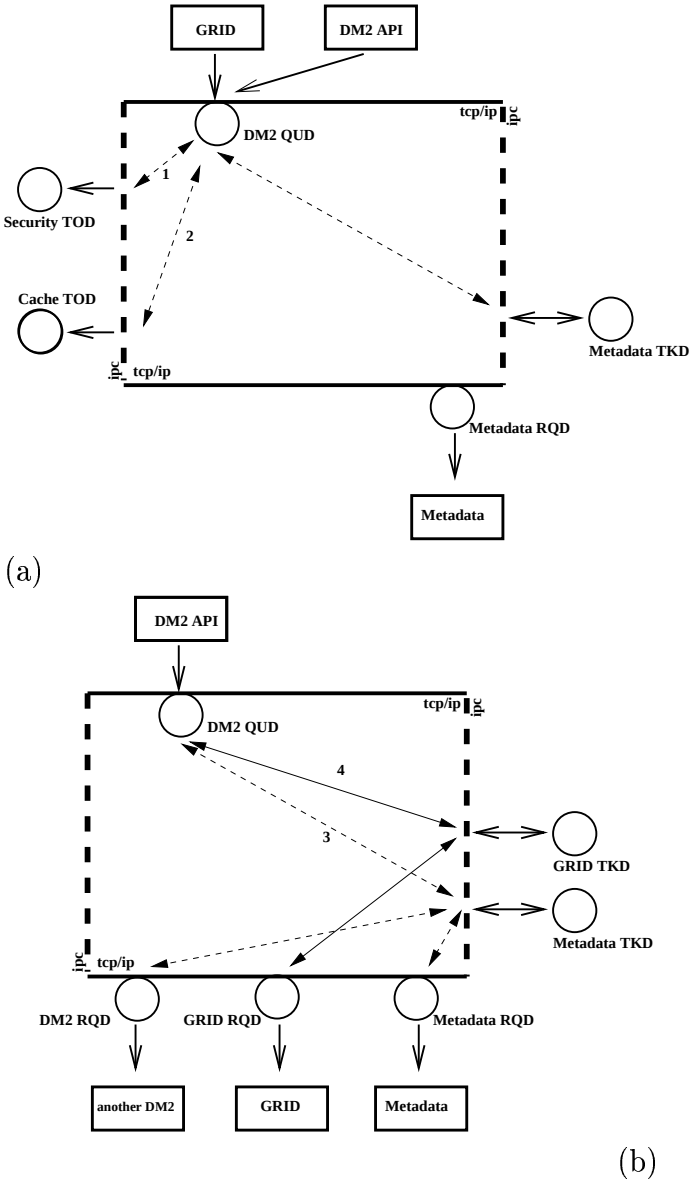


FIG. 1.4 – DM^2. Traitement d'une requête hybride.
(a) 1-Test de Securité 2- Vérification de la présence du fichier image patient dans le cache, (b) 3- Une requete distribuée est envoyee pour trouver toutes les images similaire à l'image patient, 4- le calcul des mesures de similarite est exécuté sur la grille.)

Une autre requête est alors construite et envoyée pour le calcul des mesures de similarité [28, 27] entre l'image du patient et chaque image résultant de la requête précédente (voir étape 4 de la figure 1.4b). Le service de soumission de *jobs* du middleware de la grille est utilisé pour distribuer le calcul sur des nœuds disponibles. Pour

chaque tâche lancée, le *grid replica manager* declenche un processus de recherche des fichiers nécessaires dans toute la grille (à l'aide des identifiants logiques récupérés à l'étape précédente). Si les fichiers ne s'y trouvent pas, la grille les demande au DM^2 concerné (qui peut être différent du DM^2 initial). Ce DM^2 éventuellement distant interroge son serveur local d'images DICOM, assemble les images IRM à la volée dans un buffer et renvoie les images à la grille.

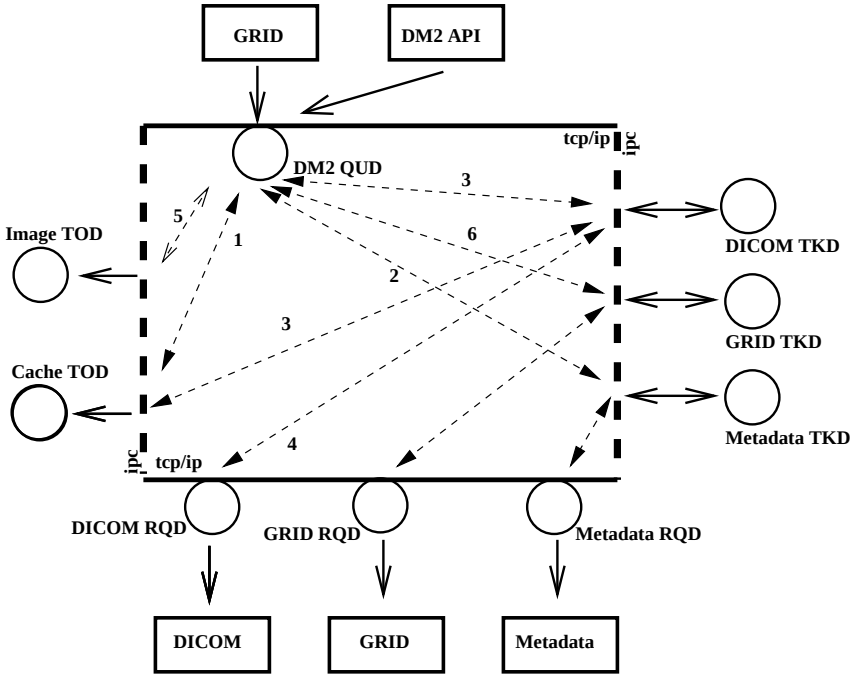


FIG. 1.5 – Exemple d'utilisation : accès à une image DICOM stockée dans un serveur PACS

La figure 1.5 détaille cette opération. En haut, le middleware de la grille déclenche une requête DM^2 pour récupérer une image : (1) Il demande tout d'abord l'image au Cache TOD. (2) Si cette image n'est pas disponible, il cherche alors dans la base de données (Metadata TKD) pour localiser les fichiers DICOM à partir desquels l'image doit être assemblée. (3) Le Cache TOD est sollicité de nouveau. (4) Si le fichier souhaité ne se trouve pas dans le cache, il doit être copié à partir du serveur DICOM. Le DICOM TKD interroge le serveur DICOM à travers le DICOM RQD. Il récupére ainsi en parallèle un ensemble de coupes DICOM (5) Les coupes DICOM sont rassemblées en une image 3D grâce à un Image TOD. (6) Enfin, l'image est stockée dans le cache (la flèche vers le Cache TOD est omise par souci de lisibilité) et est renvoyée vers la grille. Cf. étape 2 de la figure 1.4a.

Capture des images et déploiement expérimental

Une machine cliente (*engine*) de type DM^2 a été installée à l'Hôpital Cardiologique de Lyon (voir figure 1.6). Celle-ci avait la responsabilité de gérer la communication avec les imageurs IRM, de procéder à la transmission DICOM des séquences

d'images, de lancer les processus d'extraction des métadonnées et le calcul de caractéristiques des images (nouvelles métadonnées) et d'enregistrer ces informations au sein de la machine serveur (*engine*), également de type DM^2, localisée a l'INSA de Lyon, laquelle gère plusieurs hôpitaux ainsi que la communication avec la grille de calcul. De telles informations (métadonnées) peuvent être utilisées dans un proccessus ultérieur, par exemple pour une mesure de similarité (voir figure 1.7). Des tests ont été menés avec cette infrastructure et ils valident sa faisabilité opérationnelle.

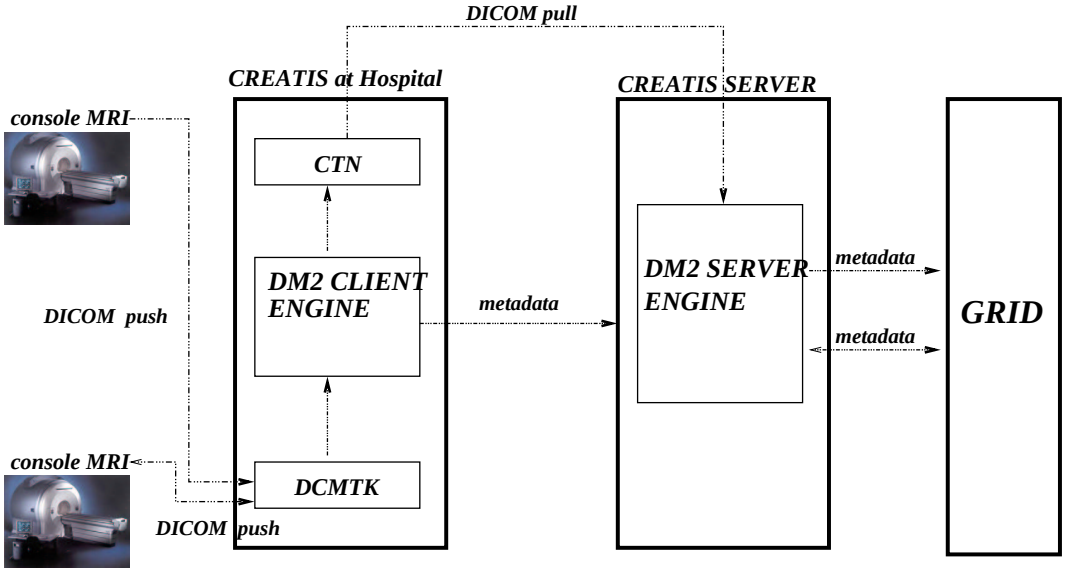


FIG. 1.6 – Hospital and Server

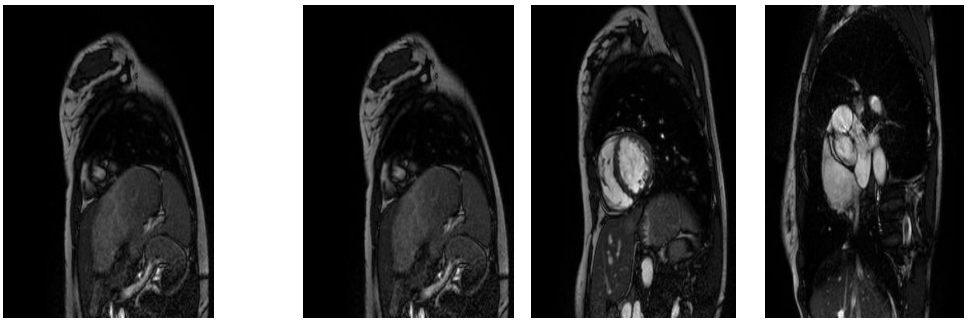


FIG. 1.7 – Mesure de similarité ; de gauche à droite : image de référence puis images similaires classées du score le plus élevé au score le plus bas. On peut noter que l'image similaire de gauche est nettement plus proche de l'image de référence que les deux autres.

1.6 Expérimentations et évaluation

Outre des tests fonctionnels(cf. section 1.5.2), nous avons mené une série de mesures de performances dont l'objectif était d'étudier la capacité du système DM^2 à gérer de grandes charges transactionelles en respectant des temps de réponse acceptables par les médecins.

Nous avons utilisé une grappe de huit ordinateurs PC équipés de processeurs à 1 Ghz et de 1 Goctets de mémoire vive. Sur la machine serveur, nous disposions de plusieurs disques durs UDMA5 ATA et IDE RAID. La vitesse du réseau était de **100Mb/s**.

Cette grappe utilisait :

```
Linux RedHat 7.3
CTN version 2.11.0 (serveur DICOM)
DCMTK version 3.52 (librairies DICOM)
PVM version 3.4.4
```

De nombreuses expérimentations ont été menées pour évaluer les performances du système. Nous ne décrirons dans cette synthèse que les tests de mesure de la capacité du système, renvoyant le lecteur au chapitre **Expérimentations** du manuscrit pour les autres expérimentations.

Nous avons, dans les tests de capacité, installé un serveur DICOM sur chaque nœud de la grappe afin de simuler huit hôpitaux. DM^2 est configuré pour pouvoir gérer jusqu'à dix sessions en parallèle avec chacun des hôpitaux. Il peut ainsi transférer en parallèle autant de coupes que de sessions définies. Cela signifie que notre serveur peut transférer jusqu'à 80 coupes DICOM en parallèle (10 coupes par hôpital).

La figure 1.8 illustre le montage expérimental.

Nous avons ainsi conçu une première expérience pour évaluer la capacité maximale du système. Nous avons donc émis une requête de lecture d'un fichier DICOM (routine *getDM2Image*) pour chacun des huit hôpitaux. Etant donné que chaque requête concerne une séquence de 10 coupes, cela implique que nos requêtes simultanées forcent le système à transferer 80 coupes DICOM en parallèle, ce qui correspond exactement à la capacité maximum configurée du système d'accès aux données.

La figure 1.9 montre le temps de réponse (hors temps de lecture disque) de chacune des huit requêtes. Il est important de souligner ici que la durée lue sur la courbe pour la dernière requête correspond à la durée totale de l'expérience. On peut ainsi constater sur la courbe que la durée de l'expérience est de 4.6 secondes ; la première requête a été entièrememt traitée en 2,9 secondes après le début de l'expérience, la deuxième en 3 secondes, la troisème en 3.3 secondes, etc.

Ces tests démontrent à la fois des temps de réponse et une extensibilité (*scalabilité*) tout à faits satisfaisants qui démontrent la performance et la faisabilité du système DM^2.

Dans une situation plus réaliste, les requêtes arrivent selon une distribution aléatoire, le système a donc l'avantage de pouvoir traiter une requête avant que la suivante arrive. Nous avons ainsi conduit une expérience au cours de laquelle 10000 requêtes successives sont séparées par un délai aléatoire dont la durée obéit à une

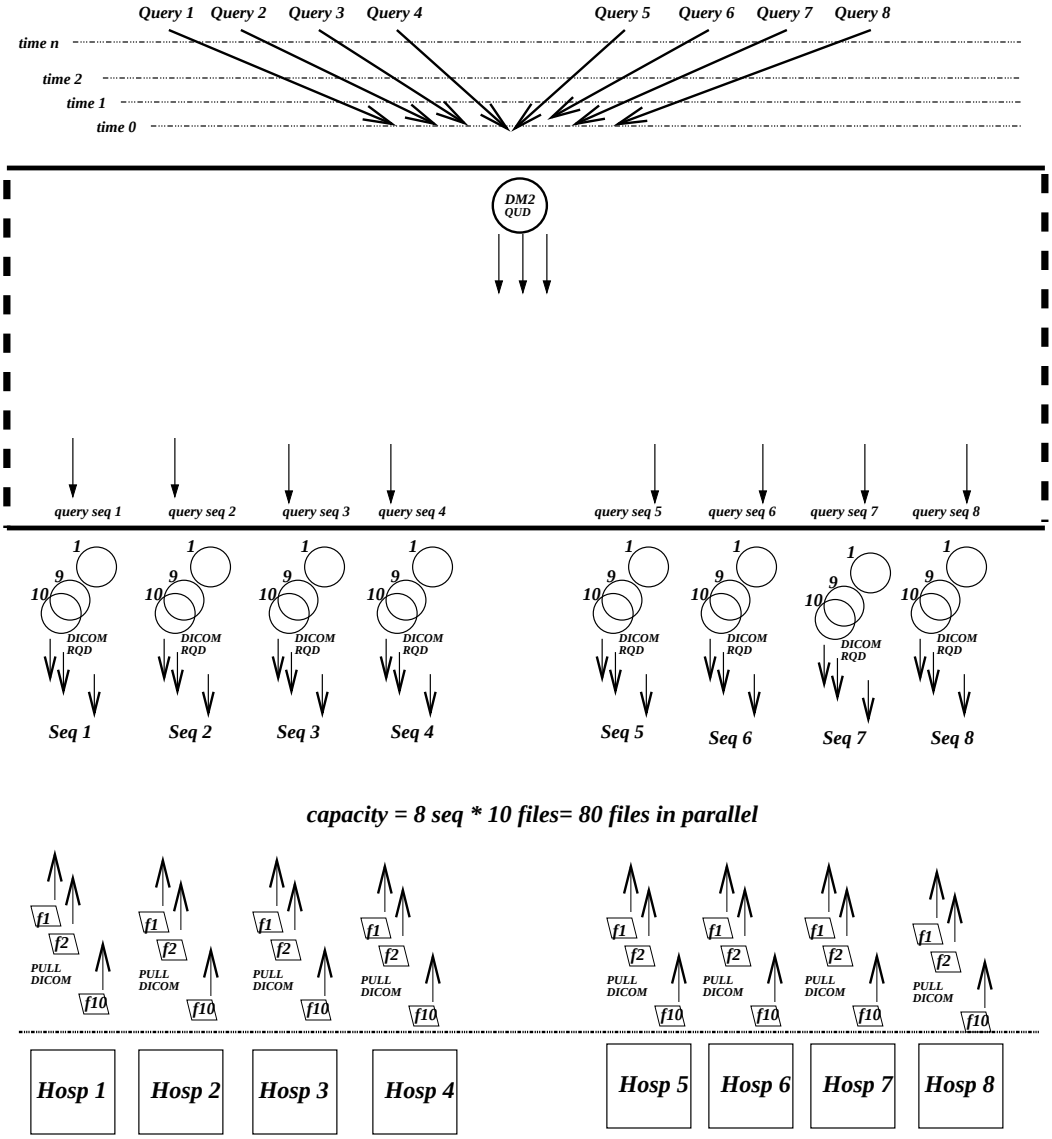


FIG. 1.8 – Saturation.

Le grand rectangle représente la machine serveur ; les petits carrés figurent les hôpitaux (serveurs DICOM). Les flèches externes en haut de la figure représentent les requêtes arrivant au même instant au driver de requêtes de DM^2 (symbolisé par un cercle). Les flèches externes en bas de la figure montrent les transferts DICOM d'une image multi-coupes au niveau de chaque hôpital. Les flèches internes représentent les messages en attente d'être traités par chacun des drivers de requêtes (RQD), qui sont symbolisés par des cercles dans la partie inférieure du serveur. Les lignes en haut représentent les instants où chaque requête arrive au serveur. Ainsi, dans le cas présenté, toutes les requêtes arrivent au même temps t0

loi de probabilité Normale ou de Poison [99] [101]). Nous avons alors d'une part mesuré le temps de reponse de chaque requête ; d'autre part, étudier le comportement du serveur DM^2, en particulier la taille de la file d'attente en fonction du temps. Les résultats (cf. chapitre **Expérimentations**) montrent que les temps de réponse restent très stables sauf dans de rares situations de saturation [4] dans lesquelles le

[4] Quand le délai aléatoire entre 2 requêtes est vraiment trop court par rapport au temps de

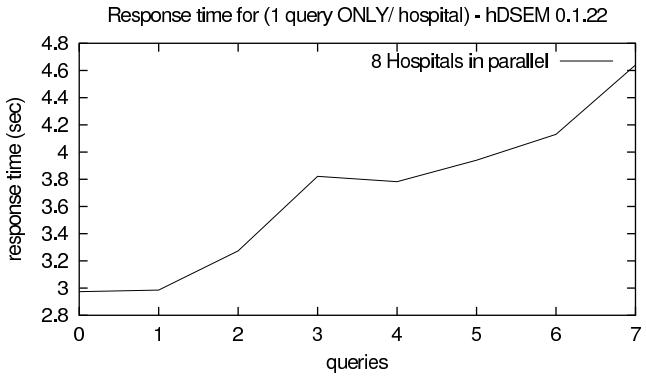


FIG. 1.9 – Saturation du système
8 hopitaux en parallèle ; une séquence de 10 images par hopital.

temps de réponse peut alors être multiplié par 2. Néanmoins, le système parvient toujours à se restabiliser dans un temps très court, démontrant ainsi sa robustesse.

En conclusion, l'ensemble des experiences montrent : (i) que le prototype DM^2 est capable de gérer de grandes charges transactionnelles et (ii) que les temps de réponse sont tout à fait compatibles avec les attentes des médecins.

1.7 Conclusion

Bien que l'application ciblée concerne les requêtes hybrides sur de grands ensembles d'images médicales distribuées, l'architecture DSE peut être utilisée pour modéliser tout système distribué exigeant performance, modularité, sécurité et partage de données.

Notre solution résulte d'un couplage entre :

- un intergiciel de grille utilisé comme fournisseur de ressources (puissance de calcul, espace de stockage...) et de fonctionnalités (authentification, réplication de données...) ;
- des entités distribuées (les DSE *engines*) chargées de l'interface entre la grille et les systèmes d'informations (médicales) décentralisés.

L'architecture DSE suggère une structuration des applications analytique (processus, transactions, *drivers*, services, interfaces) et fonctionnelle (cache, métadonnées, sécurité, traitement d'images...) qui permet à la fois de renforcer leur modularité et leur réutilisation.

Un prototype de système de recherche d'images par le contenu, DM^2, a été développé sur la base de cette architecture. Des tests fonctionelles à l'hôpital Cardiologique de Lyon et des mesures de performance ont montré la faisabilité at la performance de l'approche proposée.

Les principales perspectives à ces travaux portent sur :

- le transfert de notre prototype en un système opérationnel utilisable en milieu hospitalier. Composé d'une base intergicielle (*DSEM*) et d'une application

réponse de chaque requête.

médicale (DM^2), le prototype actuel constitue une base solide pour construire le système final.
- l'adaptation de l'architecture DSE a un modèle de grille "*datacentrique*" via notamment l'interfaçage des *drivers* DSE avec des *agents mobiles*.

Chapitre 2

Introduction

"... some scientists started dreaming. They dreamt of a way to surmount the obstacles. They dreamt of having nearly infinite storage space so they would never have to worry where to put the data. They dreamt of having nearly infinite computing power available for their institution, whenever they need it. They dreamt of being able to collaborate with distant colleagues easily and efficiently, safely sharing with them resources, data, procedures and results. And, being always worried about their research grants, they dreamt of doing all this very cheaply - maybe even for free !", ***The GridCafe*** *[173], CERN, Switzerland.*

–

Summary 2

Medical applications relate to medical image manipulation, including image production, secured image storage, and image processing. In this chapter, we show how and for which purposes medical imaging applications can be grid-enabled.

Metadata represents data about the data : in our case, the data are medical images and the metadata store relative information on the patient and hospital records, or even data about the image algorithms in use in our application platform. Metadata are either static or dynamically constructed after computations on data. We discuss how the metadata are used, produced and stored, and why a secure [109] and efficient access to medical data (and metadata) must be provided.

Currently, much work is being done around the world, in projects such as e-DIAMOND in Oxford, the European Project Mammogrid, CAMAEC in Spain, IRMA in Germany, medGIFT in Switzerland, Assert in the USA, etc. However none of these address all the problems which we have identified.

Finally, Grid technologies are not only providing additional computing and storage power, but they are also an opportunity to address new medical challenges.

–

2.1 The Challenge

One of the primary expectations of physicians regarding medical information systems is the ability to access distributed patient medical records for diagnosis and for comparison with reference annotated records. Researchers and physicians want *to query large and distributed medical images data sets by their content rather than only by their associated metadata.* The proposition of a solution to this necessity is the central objective of this thesis; it addresses different problems : (i) medical images data sets are vast and geographically distributed, so a transparent and uniform access must be provided to the user, (ii) data sets are identified and classified by associating metadata to them, so this information must be stored in distributed databases, (iii) querying an image by its content means performing image processing on the fly, so using computing resources is a must for ending the query. The huge quantity of required computing power makes desirable to use emerging technologies such as the Grids as a source of computing resources.

There are other associated problems, such as security requirements or image processing algorithms developments, but we will not consider them in this thesis.

Related work exists concerning grid middleware development, static and centralized medical images databases handling, or specialized algorithms for querying images by their content, but none of these integrate all these elements into a widely distributed environment. In this work, we aim at providing a global solution able to deal with such queries in a wide distributed environment, by using external (and cheap) computing resources (Grid). In the future, a system offering these type of queries will look like a Grid Service or Web Service.

In section 2.3 we describe a medical use case, and in section 6.3 we give a scenario for demonstrating the possible use of the prototype for a given application. This work addresses the middleware and system issues, the design and implementation itself. Our proposal to reach that goal goes through : (i) the design of a software architecture, (ii) the development of a middleware prototype for accessing the distributed medical data sources (DICOM Servers) and computing resources (Grids), and (iii) the development of software components for building a general service which instantiates image processing algorithms on an image, and shows that the application of querying images by their content becomes feasible.

2.2 Medical Data and Metadata

Medical images play a key role in medicine for diagnosis, therapy planning and treatment follow-up, and epidemiological studies. Most of the medical imaging modalities today produce digital images [23] in two dimensions (2D), three dimensions (3D) or more, like 3D temporal sequences (4D), which represent a tremendous amount of distributed data. These data must be accessed, transfered and processed in an efficient way. Handling and processing a large number of distributed image databases address several problems one must deal with :

- Huge Amounts of Data [44].
  (i) a standard 3D Computed Tomography scan (CTscan) or a Magnetic Resonance Image (MRI) represents tens to hundreds of Megabytes of data, (ii) a

single radiology department in a medium size hospital is estimated to produce tens of Terabytes (10^9 bytes) of digital images each year, (iii) the total data produced in European countries and the USA is in the order of Petabytes (10^{12} bytes).

- Distribution.
  Medical images are distributed over the medical acquisition centers throughout the territory. The necessity of automated, distributed and transparent access to remote data and processing increases day after day.
- Access Regulation.
  Although national laws concerning medical images are heterogeneous in Europe, the current trend [121] is : (i) a free access of patients to their personal medical data, and (ii) the long term archiving (from 20 to 70 years) of all medical data for pathologies and epidemiology studies. Medical data are sensitive and should only be accessible by accredited users, which makes data manipulation over a wide area network difficult. Users often want to associate additional data (metadata) to the original data or images, such as clinical features, personal observations, biology or cythology results gathered by medical experts. Although patient metadata are the most sensitive part of the data, no medical data, including the image content, should ever be accessible to unauthorized users.
- Computing Requirements.
  Automated medical image analysis and processing tools have been developed in computer science and signal processing laboratories for more than 15 years. Beyond the low level processing for signal filtering or 2D/3D reconstruction, medical image processing algorithms [111] proved to be useful for image enhancing, visualization, comparison, quantitative evaluation, and various simulation processes. These algorithms provide diagnosis assistance, therapy planning tools, and a way of performing both tedious image analysis tasks that are not tractable by humans for very large datasets, and also advanced imaging tools for high level modeling.

The medical images related to a patient, are not self consistent in the sense that the physician needs to interpret them in a global context. The images content is only relevant for medical decisions when additional parameters such as patient age and sex, complementary records and/or sociological and environmental conditions are considered. Beyond simple diagnosis, many other medical applications are concerned with the data semantics and require rich metadata content. Epidemiology, for instance, requires the study of large data sets and the search of similarities between medical cases (are all affected patients belonging to the same category ? Same age range ? Same professional context ? etc). There is a need to take into account the metadata about each medical case, like features resulting from images processing (*e.g,* pathologies or anatomical shape descriptors) in the similarity criterion. Therefore, medical metadata carrying additional information on the images are mandatory.

Although weakly structured, medical data have a strong semantic content and metadata is necessary to describe it. Medical data are not "raw data", like in many grid applications (physics, astronomy, environment). They are semantically rich data that are even enriched by the use of metadata. These structured metadata permit

a better handling of the data itself, providing indexation algorithms, information retrieval, data classification, and data caching, in which the relevant meaning of the data is considered for improving data management. The current interest in semantic grids and metadata management at the Global Grid Forum [162] is an indicator of this current trend.

Although there is no universal standard for storing medical images, the most established industrial standard is DICOM (Digital Image and COmmunication in Medicine) [142]. DICOM describes both an image format and a client/server protocol to store and retrieve images on/from a medical image server. Most recent image acquisition devices implement the DICOM protocol. The DICOM file format is made of a header containing metadata followed by one or several image slice(s) in a single file *(sequence of images)* ; While a 2D image is just one slice, and is stored as a single file, a volume (3D) or temporal (4D) image might be represented as several slices. This means that it is possible to store the sequence of images as a sequence of files, where each one has its own associated metadata (see figure 2.1) .

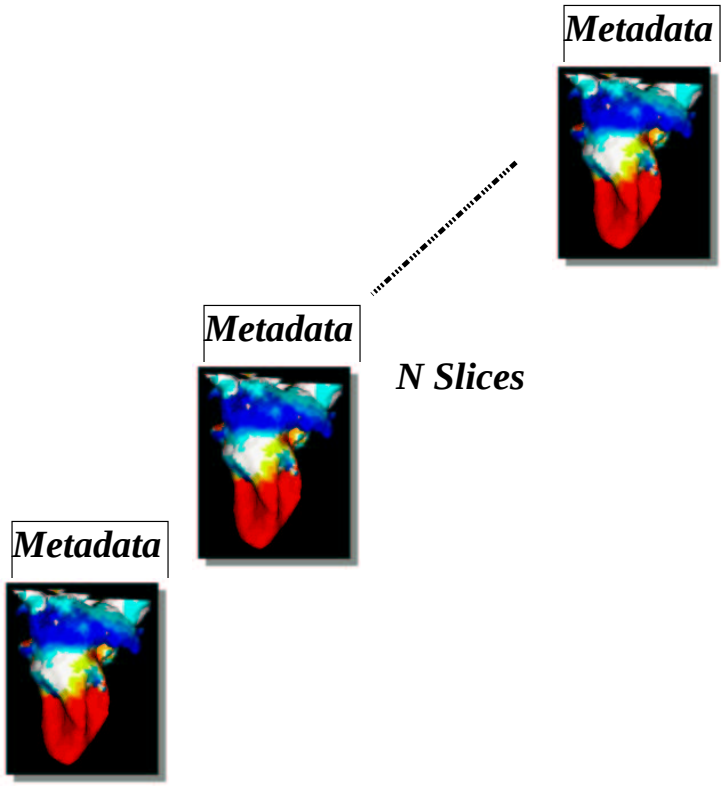


FIG. 2.1 – Sequence of DICOM Images.
Metadata for each slice in the sequence (N files).

DICOM images already contain several acquisition-related metadata in the image header. Precisely, there are two kinds of information :

- sensitive **patient-related** metadata such as patient name, sex, age, radiologist name, hospital, etc.
- **image-related** metadata such as image acquisition device type, constructor name, acquisition conditions, acquisition date, number of images stored, size

of images, field of view, etc.

However, in practice, these in-file metadata are often incomplete (the patient name may appear but not its age, etc) and not well adapted for data search and query. Therefore, DICOM servers usually extract the in-file metadata and store them in databases where they can be completed and exploited.

2.3 Medical Use Case

To illustrate our goals we will consider the following example : *a cardiologist looks for cardiac images similar to those acquired on one of his patients to confirm his diagnosis. He wants to rank the images through a similarity score computed between them and the patient image. He is specially interested in cases with an ejection fraction (EF)* [1] *greater than 0.5. Once the images are ranked, he needs to visualize the most similar cases and their attached diagnoses. His own diagnosis can then be stored in the patient record and added to the information system.* In technical terms, the cardiologist needs to :

1. Query a distributed metadata database holding information about images. These metadata might be stored with each image they are attached to.
2. Access a large data set with comparable cardiac images distributed over different hospitals.
3. Take advantage of pre-computed metadata, stored in the database, and useful for reducing the query domain (the EF).
4. Make computations (similarity measurements) on a large number of images in a very limited time.
5. Update the metadata database and its associated medical image data collection.

The first two items are related to the problem of *accessing medical data*, the 3rd item is about *images indexing and pre-computing*, the 4th one has a relationship with *content-based image queries and hybrid queries*, and the last one is about *tracking* (see below).

We briefly describe each item as follows :

– Accessing medical data.
  A specificity of medical data is their strong semantic content. A medical image itself has often low interest, in itself, if it is not related to a context (patient medical record, similar cases [44] [110]). In a first step, users first query the metadata database to identify relevant information, *e.g,* the cardiac images of the concerned patient. Then the application will query a data location service to get the physical location of all the requested images. That might be spread over several hospitals. The application will retrieve the set of DICOM images from the selected hospitals and assembles them in a single 3D image (one file) or in temporal sequences, that will be returned to the cardiologist for visualization. Clearly, the latency for accessing those data must be minimized.

[1] The ejection fraction is the amount of blood that the left ventricle pomps out per beat into the body when it contracts.

– Images indexing and pre-computing.
  The previous query implies accessing medical databases and using complex access patterns involving both metadata related to each patient record and image content analysis. In order to improve the response time, it is desirable to pre-process the images and to generate metadata indexes useful for image retrieval, such as histograms, texture parameters, etc, considered as metadata here.
  Other metadata include patient related information (name, age, ...), diagnosis-related, and therapy-related information.
  A good pre-computing strategy can reduce the data search domain and decrease the query time.
– Content-based image and hybrid queries.

  The medical application must analyze on-the-fly images when a query arrives, in order to extract features from the image when they have not been pre-computed (e.g, similarity scores). This kind of query, which performs computations for extracting features from the image before returning results to the user or taking a decision, is understood as a *content-based* search.
  In our example, the query needs to : (i) query classical metadata (cardiac images, physician name, and location), (ii) query pre-computed metadata in a database, (iii) access remote raw data (DICOM images), and (iv) compute on the fly additional information from the images, before deciding if it is helpful or not.
  Such a complex query is called a *hybrid query* as it involves both metadata and content processing. Hybrid queries can obviously take advantage of images indexing for reducing the search domain. Content-based access to data by visual features on the image content is complementary to text-based queries on the metadata, and is unlikely to ever replace them completely. That is why hybrid queries are needed.
– Tracking.
  In order to improve the diagnosis a physician ask for additional computations that produce new processed images. However, the physician always needs to come back to the raw data when visually analyzing a processed image. Conversely, for each input data it is of interest to optimize computations to record the results of previous image processing algorithms. Thus, relationships must be established between the raw data images, the computed images, and the additional metadata. It should always be possible to know, for a given image, where it originates from (which algorithm and which input image(s) were used to produce it). This requires updating the (distributed) patient and image database in order to track images and metadata, with new images and new metadata. This *tracking* represents the history of a patient.

2.4 Distribution and Grids

Medical data are stored and archived inside each medical image producer site (hospitals, clinics, radiology centers). The medical record (image files and metadata) of one patient is therefore distributed over the different medical centers that have been involved in the patient health care. Several Picture Archiving and Communication Systems (PACS) [25], Radiology Information Systems (RIS) and Hospital Information Systems (HIS) have been developed to provide data management, visualization, and, to some extent, data processing [2].

Even if a standard for health-care specific data exchange such as HL7 [197] exists [3], at the moment it is not widely used in solutions including integrated PACS, RIS and HIS. Moreover, PACS and RIS are usually designed to handle information between a single hospital. They do not consider either the transmission of such sensitive data between sites, nor the integration with external computing or storing facilities.

With the performances of computer networks and hardware technology, parallel and distributed processing have become a key technology which play an important role in determining future research and development activities in many academic, research and industrial branches.

However, medical applications need more than distribution, they require obtaining access to specialized resources, such as those a Grid can offer. As defined in [158], a Grid *"is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements"*. These types of resources are highly desirable for medical systems.

Grid technologies, as a data intensive manipulation framework, are promising for medical image management [30] [32]. They offer large scale and distributed storage capabilities associated with a better use of computing power. They permit the sharing of data and resources which is important for clinical practice since hospitals and clinics usually do not own much computing power. Beyond the obvious interest of grids for clinical practice, this technology favors research by allowing scientists to share datasets and image processing algorithms more easily than ever. All of these facts have risen the awareness of the benefits of grid technology in the medical community recently.

Grid Computing research works have long been focused on the efficient use of computing resources in terms of scheduling, resources discovery and usage, in projects like Globus [14], Condor [127] or Legion [15] [128]. Interest in data being manipulated by grids only grew when the amount of data used or produced in grid applications started to become a real problem. In this context, the two European projects, DataGrid [126] and EGEE [143], both consider huge databases handling

[2] PACS archive images and allow image transfers. RIS contain full medical records : image-related metadata and additional information on the patient history, pathology follow-up, etc.

[3] HL7 (Health Level 7) : An ANSI standard for health-care specific data exchange between computer applications. Its name makes reference to the top layer (Level 7) of the Open Systems Interconnection (OSI) layer protocol for the health environment.

and fast distributed computing, since petabytes of raw data provided by applications such as particle physics experiments, earth simulation and bio informatics, are supposed to be delivered to the grid.

A big concern when distributing medical data over a grid is privacy. Medical data are confidential and should only be accessible to the patient himself (herself), the medical team involved in his (her) health care, and, under some restrictions, for research purposes. Therefore, a medical grid, opened to a wide community of users, must enforce strict access right control. The lack of data security integration is today a major weakness of emerging grids middlewares to address medical requirements.

In this thesis, we address the topics of data access and manipulation in a medical grid, but not the security issue. It is out of our scope of interest ; however it is a matter of research by our team [187].

2.5 Ongoing Work

In this work we address both computer systems and middleware topics (low level), and dedicated medical images applications (high level). In this section we describe the ongoing work in the **medical application field** [112]. We selected the most significant projects in our field of interest.

Our thesis is part of the *MediGrid* [131] project, a French ACI project which aims at exploring the use of Grid technologies for processing huge medical image databases, co-operating and interfacing with the European DataGrid project [126] and EGEE project [143] (see section 3.1.2). It is also underlined in the *Ragtime Project* [195], another French project [4] which aims at supplying Grid middleware tools in order to provide access to huge medical image data-sets.

The *Medigrid Project* addresses several topics, such as (i) an MRI images simulator, (ii) a 3D anatomic and functional cardiac model, (iii) the design and development of mechanisms for accessing medical images in a distributed and heterogeneous environment, and (iv) the execution of hybrid content based and indexed queries of medical images.

The *Ragtime Project* addresses : (i) middleware issues for getting access to data, (ii) definition of distributed databases for managing target medical metadata, and (iii) development of mechanisms to connect medical data with a Grid, and (iv) development of tools for image visualization and data manipulation. Ragtime also addresses the development of applications as the construction of a statistical atlas of human organs, the automatic analysis of mammographies for cancer diagnosis [107] [108], modeling of cardiac dynamics, and the proteins analysis.

This work contributes [45] [44] [51] to items *iii and iv* of *Medigrid* and items *i, ii and iii* of *Ragtime*.

2.5.1 Data Grids Projects

In this section we describe projects which are based on the existence of large databases and use *Data Grids* infrastructures.

[4] Supported by the region Rhone-Alpes

Mammogrid

The Mammogrid Project [5] aims at developing a European-wide database of mammograms for investigating important health-care applications as well as cooperating with health-care professionals throughout the EU [29]. The project looks for helping in *breast cancer diagnosis and treatment.*

Mammogrid intends to use Grid infrastructure in order to enable distributed computing at a European scale. Most important applications to be implemented address two main problems : (i) Image variability, due to differences in the acquisition processes , and (ii) Population variability, which causes regional differences and affects criterions for treatment of breast cancer. The project intends to pave the way for potential knowledge discovery in the diagnosis and understanding of breast cancer [146].

Of primary importance is the security of the patient raw data, that must remain anonymous and confidential, as well as the associated records. This implies the use of efficient information structures for dealing with data integrity, quality and consistency.

The MammoGrid database manages medical images (such as MRIs) associated with the patient records, thus providing to the health-care professionals a platform for further clinical studies.

Let us note some technical points :

- The Mammogrid database contains series of individual images (*e.g.,* MRI) and copes with the *Standard Mammogram Format (SMF)* [147]. It is also DICOM compliant although the source image files are not in that format. They must be converted into SMF or DICOM format before becoming available.
- Its database is built from multiple federated databases [32] including X-ray mammograms, MRI, and metadata, meaning that raw data must be transfered to a *data store* [29].
- The database is also built using the SMF standard, which means converting files into that standard.

e-DIAMOND

Oxford University's eDIAMOND [144] grid computing project [6] is part of the United Kingdom's e-Science [145] [7] program, a nationwide initiative to make access to computing power, scientific data repositories and experimental facilities as easy as the Web makes access to information.

In the UK there are about 40.000 women diagnosed with breast cancer every year, and eDiamond is aimed at helping physicians to make more accurate diagnosis of breast cancer. The project uses a large database of mammograms, to provide image

[5] Supported by the Commission of the European Union Information Societies Technology (IST) Program

[6] Oxford University led eDIAMOND.

[7] **e-Science** is a UK programme which aims at developing and brokering generic technology solutions and generic middleware to enable e-Science and forming the basis for new commercial e-business software. The e-Science Center (OeSC) at Oxford University is involved in many collaborative e-Science projects, in different scientific fields such as : (i) physics and engineering, (ii) health, (iii) biological and environment, and (iv) grid technology.

comparisons based on diagnostically meaningful information, overriding the variability observed during the image acquisition process. The system aims at defining quality controls in screening programs and in the study of breast cancer epidemiology.

The project pools and distributes information on breast cancer treatment, enables early screening and diagnosis, and provides medical professionals with tools and information to treat the disease. It gives patients, physicians and hospitals fast access to a vast database of digital mammograms. It is also expected to help reducing the rate of false-positive diagnosis.

Once the patient's mammograms are loaded into the system, a software screens them for abnormalities by comparing current mammograms with anterior ones from the same patient. Physicians can also visually compare similar cases extracted from the database [31]. Meanwhile, data-mining and image processing techniques explore the stored mammograms in order to help in both the detection of abnormal features as well as the diagnosis of cancers.

Standardization of images from different centers in the UK would *"enable a database to be built using scans taken on different machines or at different sites, with the effect being that they would all appear as if they were produced on the same machine under the same conditions"*[144]. The facility to standardize digital mammogram images (SMF) [147] enables comparison of mammograms in terms of intrinsic tissue properties independently of scanner settings, in the hope to help radiologists to compare and evaluate mammography scans, no matter where or when they were created.

2.5.2 Other Projects

The projects described in this section require for computing power and have the potential to make use of *Computing Grids* infrastructures; however, they are not computing grids projects.

DISMEDI

DISMEDI, the Distributed High Performance Processing of Medical Images System, is a project of the High Performance Networking and Computing Group of the Polytechnic University of Valencia [20] (Spain) [151]. It is a project common to the Polytechnic University of Valencia [151] and hospitals Malva-Rosa and de la Ribera in Spain.

This project aims at designing [150] a high-performance, low-cost, medical server offering image-processing software for 3D segmentation, 3D reconstruction, navigation, etc. The system works in a standard DICOM network as a DICOM client/server and is oriented to the domain of Digital Radiology. It is a system for (remote) image diagnosis, but the final product resembles more to an Advanced PACS than to an image processing tool giving access to huge distributed databases.

CAMAEC

The *CAMAEC* project is also a project of the High Performance Networking and Computing Group of the Polytechnic University of Valencia (Spain) [151]. Its main goal is to build a complete parallel computing system [152] for the simulation of action potential propagation in a two-dimensional cardiac tissue using a cost-effective cluster of PCs. CAMAEC is a specific parallelized system [33], which uses high performance computing techniques and tools (MPI).

medGIFT and CasImage

The Division of Medical Informatics at the University Hospitals of Geneva [155] works intensively in PACS related research, distribution of medical images, advance image processing, content-based image retrieval, etc.

medGIFT [198] aims at developing new tools for content-based image retrieval (CBIR) and Content-Based Visual Information Retrieval (CBVIR). The *CasImage* [199] program is a collection of medical cases dedicated to pulmonary diseases including medical lung high-resolution CTs images (DICOM). It is a *public reference image database* which is useful for establishing diagnosis.

PTM3D

In Paris-Sud University, researchers are developing the system *PTM3D* [200] for visualizing, navigating through, and analyzing [30] multi-modal medical image data sets as CT, NRI, US. However, it does not consider distributed databases for doing hybrid queries.

AQUATICS

AQUATICS [154] is a project of the EUTIST-M Initiative [153] which aims at providing a simple and effective collaborative diagnosis and treatment planning to interventional radiologists performing endovascular repair of Abdominal Aortic Aneurysms. It delegates sophisticated IT tasks, such as medical imaging processing, to a specialized distributed service based on lightweight middleware. This project will address grid computing, but does not consider hybrid queries processing.

[20] High Performance Networking and Computing Group of the Polytechnic University of Valencia [151] is very active in High Performance and Networking applied to medical images and information. They are also involved in the EUTIST-M Initiative [153] which researches and develops in the Medical Sector by grouping a consortium of 40 European Companies, Universities and Technology Centers for improving medical information solutions. The application areas are Radiology, Orthopedics, Oncology, Intensive Care Units, Surgery, Dermatology.

Recently, this group has developed a middleware for storing, retrieving and processing DICOM medical images [98]. That middleware aims at providing transparent access to medical imaging resources. The application built on top is related with volume rendering of tomographic studies and not considers queries by-content

IRMA

Another CBIR project is *IRMA*[156] at the Aachen University of Technology (RWTH Aachen) in Germany. It aims at developing and implementing high-level methods for content-based image retrieval with prototypical application to medico-diagnostic tasks on a radiological image archive. The objective is to perform semantic and formalized queries on a medical image database, which includes intra- and inter-individual variance and diseases. Its main focus is in the content-based search algorithms and not in the computer mechanisms for applying one algorithm in a general sense.

Assert

the *Assert* project [157] of the Department of Radiology at Indiana University and the School of Medicine at the University of Wisconsin (USA) uses a database that consists of a High Resolution Computed Tomography of the lung. To pose a query to the database, the physician circles one or more pathology bearing regions (PBR) in the query image. The system then retrieves the n most visually similar images from the database using an index comprised of a combination of localized features of the PBRs and of the global image.

2.5.3 Comments

The Mammogrid Project aims at developing a European-wide database of mammograms by using multiple federated databases. This approach means that the raw data must be transfered to a *data store*. Our understanding of the problem motivates us to access the raw data at its origin (hospitals) without exchanging files. We consider only a federated metadata database by geographical region, not a raw image data collection. Additionally, Mammogrid requires for file conversion into the SMF standard. We aim at using the image data as it is in the origin (DICOM or another format). While Mammogrid works with images, we must deal with temporal sequences of images (4D).

eDIAMOND Project aims at defining quality controls in screening programs and in the study of breast cancer epidemiology. Its direction has some important differences from ours :

- This project looks for federating data and building a huge standardized database of images, while we want to ease access into the existing data without moving the master images, which means taking advantage of distributed data at the origin.
- E-Diamond provides grid computing resources within its own environment whereas we aim at developing systems which use existing grid resources.
- Contrary to Mammogrid or eDIAMOND projects which handle series of individual static 2D images, we must manage 3D volumes and temporal sequences of images, which means huge amounts of complex structured objects to deal with.
- We address the generic problem of medical image data management whatever the application field.

Projects as the ones described in section 2.5.2, address some of the problems which we must deal with. However, they are different than ours, in the sense that they are not oriented Grid projects, even if in the future they can fall in this research direction. For example, **DISMEDI** has some elements of image processing (remote image diagnosis), but it is closer to a PACS system than to a huge distributed system capable of executing image processing. **CAMAEC** implements image processing over 2D cardiac tissue images rather than temporal sequences (4D), as we do. DISMEDI and CAMAEC are cluster oriented projects rather than Grid Computing projects, as is ours.

Some projects address the problem of querying by the content **(medGIFT, IRMA, Assert)**, but not of the size that we have to deal with, and without using external computing resources (Grid) as we do.

Similar to the **IRMA** project, the **Assert** project is based on content-based image retrieval methods, whereas we are interested in the computing methods for easing and accelerating a query which uses the same algorithms. We are not looking for content-based query algorithms, we aim at allowing a scientist to apply different existing algorithms (or his/her algorithms), in a massive way, over a huge database. Our project will allow systems such as IRMA or ASSERT to run their algorithms taking advantage of Grid Computing Resources and large distributed databases of medical images.

AQUATICS addresses middleware issues but does not study hybrid queries processing. However, some image processing, simulation, and modeling algorithms are very CPU intensive and need a parallel implementation in order to get executed in a reasonable amount of time compatible with clinical practice constraints. In order to deal with medical images, we have to split raw image data from metadata [45] for allowing the selection of images for a posterior process.

2.6 Positioning

Computational grids [5, 12] have encountered a large success among the distributed computing community. Many technical developments aim at providing a middleware layer for submitting remote jobs [41, 4, 13], storing data [6], or monitoring a distributed system [40]. But most importantly, from the user point of view, grids should provide transparent access to distributed resources and ease data and algorithms sharing.

Grids make the promise of large computing power and data storage space, but more benefits are expected in the medical imaging domain beyond these capabilities. Indeed, grids are a vector for permitting the creation of large scale distributed datasets, enforcing the use of common standards, and permitting the medical communities to share computing resources and algorithms. Grids are likely to have a deep impact on health related applications by playing a key federative role [24]. They will provide a logical extension to regional health networks [25] by allowing distant sites to collaborate and exchange their data for specific research purposes.

Sharing data sources will facilitate research on pathologies and epidemiology. Connecting distributed data sources will allow researchers to assemble *virtual data*

sets suited to statistics extraction or to study rare diseases. With a proper grid infrastructure, experiments can be led at a scale never reached before. Sharing resources will facilitate the access of health centers to image processing services even though they might involve computation. Sharing algorithms will ease the access to such image processing tools for the end user.

In this context, we are interested in the *integration process of tools and services.* We propose to explore distributed architectures and middleware techniques in order to :

- provide the Grid an access to medical images and data which are *external* to the Grid environment. This enables a Grid to work with data which are not in its domain [8].
- ease the process of hybrid queries over huge distributed medical images databases.
- take advantage of grid computing for accelerating massive hybrid queries.

To remain transparent from the user point of view, a middleware layer should take into account the particular needs of medical applications. We look at dealing with different kinds of medical images; however we are working towards having an initial prototype which allows the management of temporal cardiac sequences of images (sequences of volumes 3D). This fulfills the research interests between CREATIS laboratory [148] and LIRIS laboratory [149].

In this work we propose an architecture which is called (**Distributed Systems Engines - DSE**), and defines a multi-layer structure which is implemented as a prototype in two big chunks, the middleware or manager, called **DSEM**, and the application, called **Distributed Medical Data Manager ($\mathbf{DM^2}$)**. These concepts are developed in chapters 4 and 5.

2.7 Document Overview

This document begins with a survey of related work in the fields of distributed systems, grids technologies, massive storage, middleware and image storage. The third chapter provides an overview of the proposed architecture *(DSE)* and explores its advantages in terms of extensibility, scalability and integration. Then, chapter 5 shows an implementation of our architecture at two general levels : middleware *(DSEM)* and application *(DM2)*. The application is developed for managing medical and image data.

The first part of the chapter 6 details experiments which analyze the behavior of our prototype when responding to stress conditions, and the second part addresses the use of the system by showing an user application. The next chapter (7) discusses our work in comparison with the open problems and the state of the art. It concludes this manuscript and traces further work.

Finally, special terminology and acronyms are detailed, as well as two annexes (8, 9) which describe useful information about a real test environment, and the

[8] At the moment a Grid works with data which it totally controls, e.g, which are registered and stored in the Grid. Moreover, medical images are *de facto* stored in external systems and can only be partially (once anonimyzed and encrypted) transfered to the Grid.

query-by-content algorithms which were used in the prototype application.

Chapitre 3

Related Work

*"In a **Metacomputer**, the resources are usually connected within the LAN belonging to one organization, and its coupling is restricted to sharing storage, files and separate applications, ... in the **Grid** the coupling goes both much deeper and wider, the resources are often distributed on the WAN and they belong to and are administered by different organizations", **DataGrid Project Report, EU** [19]*

*"**Cluster Computing** is about resources aggregation in a single administrative domain, ... **Grid** computing is about resource sharing and aggregation across multiple domains.", **Rajkumar Buyya**, University of Melbourne, Australia*

*"**Grid** systems integrate resources that are more powerful, more diverse, and better connected that the typical **peer-to-peer** resource, ... these are both concerned with the pooling and coordinated use of resources within distributed communities, but are based in different communities and focus on different requirements.", **Ian Foster**, Argonne National Laboratory, USA*

–

Summary 3

We aim at offering image querying by the content and hybrid queries over a huge set of partially distributed medical images. Additionally, the system has constraints of high-performance, high-throughput and data-intensive computing. Thus, our project has some relationship with different fields of Computer Science.

Due to the widely distributed environment of the project, we make a descriptive revision of on going work in Distributed Computing. The high-performance, high-throughput and data-intensive constraints have motivated us to take a look at Mass Storage Systems and Grid Computing fields.

***Grid Computing** is addressing the problems of resource sharing, coordinated problem solving, and dynamic, multi-institutional virtual organizations, while other Distributed Technologies deal with aggregation of resources in a single administrative domain. The key is the coordinated use of resources.*

*Grids are classified as : computational, scavenging and data grids. A forth kind of Grid has been presented : the **data-centric Grid**. Characteristics of on-line data increasingly suggest that they should be used in place, rather than copied around. Datacentric grids have been proposed in order to deal with problems which the code goes to the data, instead of data going to the code.*

We discuss in this chapter the most important projects in Grid Middleware : Globus, Condor, Legion, Unicore, Gridbus, SRB, and also many Grids Related Projects such as DataGrid, EGEE, EuroGrid, K-GRID, NASA IPG and CrossGrid.

*We them compare grid and **Peer-to-Peer** systems in terms of relevance to our concerns.*

Object computing infrastructures such as CORBA, DCOM and Java-RMI are also presented. Indeed thus define interoperability between distributed objects in a heterogeneous, distributed environment and in a way transparent to the programmer, which interferes with some of our concerns.

*We present also other technologies of distributed systems, such as **Distributed File Systems and Mass Storage Systems**. A Hierarchical Storage Manager (HSM) is a kind of MSS. Some Mass Sto-*

rage Systems (HSM/MSS) are described (Castor, Enstore, Eurostore, HPSS); their commonality being that all address the need for storing vast quantities of data (hundreds of TB to PB scale).

Medical Information and Medical Images are stored in ***PACS*** *systems. A PACS should allow the functional ability not only to distribute images to the requestor but also to communicate to physicians for patient scheduling, download patient demographics to modalities, to track image location, and to assemble collaborative material necessary to interpret the image.*

The Digital Image and COmmunication in Medicine ***DICOM*** *facilitates interoperability of medical imaging equipment by specifying protocols for medical devices, syntax and semantics of commands and associated data for transmission information. It also facilitates operations in a networked environment without the requirement for Networked Interface Units.*

Most PACS systems are built on top of the DICOM standard [49]. Though our research is not directly concerned with PACS we highly present the DICOM standard and PACS systems as our system managers DICOM compliant.

As was described in chapter 2 section 2.1, we aim at offering medical image querying by the content and hybrid queries, in a partially distributed environment with constraints of high-throughput computing. We need to deal with huge quantities of image data sets which require vast resources of computing and storage. Thus, our project has some relationship with various fields of Computer Science : distributed computing, grid computing, (image) data storage.

In section 3.1 we discuss grid technologies and, more precisely, work in grid middleware. Then, in section 3.2 we take a look at other distributed computing frameworks. Finally, section 3.3 is dedicated to image storage.

3.1 Grid Technologies

The challenge of grid computing is the integration of heterogeneous computing and data resources aiming at providing a global computing space. The achievement of such a goal will involve revolutionary changes in the computer sciences, by enabling large scale resource-sharing across networks.

Grids enable the sharing, selection, and aggregation of a wide variety of resources including supercomputers, storage systems, data sources, and specialized devices [16] that are geographically distributed and owned by different organizations. This technology have led to the possibility of using wide-area distributed computers for solving large-scale and data-intensive problems [8] [9] . It provides consistent, pervasive, dependable and transparent access to the managed resources [52].

Grid applications, which are often multi-disciplinary and large-scale processing applications, often couple resources that cannot be replicated at a single site, or must be globally distributed for practical reasons. In this way, the Grid allows users to solve larger-scale problems by pooling together resources that could not be coupled easily before. A Grid is not only a computing infrastructure [11], for large applications, it is also a technology that can bind and unify remote and diverse distributed [54] resources.

In order to allow a secure and robust infrastructure to be built, software standards (OGSA [137] and WS-RF [163]) are being defined and tools such as those provided by the Globus Toolkit [129] offer the necessary framework. We do not aim at improving these standards; but aim that our project must be easily adaptable on top of standard interfaces.

Often, grids are categorized by the type of solutions that they best address. The three primary types [158] of grids are (i) Computational grid, (ii) Scavenging grid, and (iii) Data grid. Of course, there are no hard boundaries between these types and often grids combine two or more of these [56] types. However, as someone considers developing applications that may run in a grid environment, he (she) must appoint the type of grid environment that will be used in order to make the appropriate decisions.

- **Computational Grids** are focused on setting aside resources specifically for computing power. They usually use high-performance servers.
- **Scavenging Grids** offer large numbers of desktop machines which can be scavenged for available CPU cycles and other resources. Owners usually give

up control over their available resources in order to participate in the grid.
- **Data Grids** house and provide access to data across multiple organizations. They are responsible for tackling and managing large amounts of data in geographically distributed environments, allow one to share data, and also manage security issues, such as who has access to what data.

We are not interested in *Scavenging Grids* ; we interact principally with *Computing and Data Grids*. In our project we neither propose, nor develop any Grid middleware. Rather, we propose a system which can be interfaced for a *Computing Grid or a Data Grid* in order to have access to remote and secured medical data. At the moment, existing Grids work with data stored in the Grid, which means some defined storage space is managed in the internals of the Grid, for example Storage Elements (SE) [126]. However, most of the time, the medical data cannot be permanently stored into the Grid. In this way, we place our work as cooperative link between a Grid and specialized medical systems that gives a Grid the ability to work with external medical data.

Another common distributed computing model, that is often associated with or confused with Grid computing, is *Peer-to-Peer computing (P2P)* . In fact, some consider P2P as another form of Grid computing [53] [55], although there are important differences [18] (see the section 3.2.2). We do not implement P2P techniques in our project, but it is a way of implementing and interconnecting *the engines* proposed in the next chapter.

On a Grid, the underlying infrastructure for distributed storage and computation is hidden so that the users do not have to know where and how the data are stored or the applications run ; in fact, it works as a shared networked computer resource [8] [9] . A Grid manages data and computation intensive problems for which distributed computing and resource sharing offer a solution. Thus, middleware tools have to be build in order to implement this computational and storage infrastructure.

In practice, a Grid's middleware should allow applications to simultaneously use large numbers of resources, complex communication structures, resources from multiple administrative domains to cope with, stringent performance requirements, and to make dynamic resource requests [10] [16].

3.1.1 Grid Middleware

Globus

The Globus project [129] is an American led multi-institutional [1] research effort that intends to enable the construction of computational Grids and focuses on enabling the application of Grid concepts to scientific and engineering computing. It develops fundamental technologies and prototype software tools (the Globus Toolkit)

[1] Globus started in 1996, the main collaborators are Argonne National Laboratory's Mathematics and Computer Science Division (Ian Foster) and the University of Southern California's Information Sciences Institute (Carl Kesselman). Other institutions as NASA, Universities of Chicago and Wisconsin, DARPA and the National Science Foundation (NSF, National Technology Grid) also contribute in its development.

which can be used to build computational grids on a variety of platforms [2].

Globus also does basic research in resource management, data management and access, application development environments, resource location, communications, and security. It supports the construction of Grid infrastructures spanning supercomputer centers, data centers, and scientific organizations.

A central element of the Globus system is the Globus Toolkit [14], which defines the basic services and capabilities required for constructing computational Grids. It provides a software infrastructure that enables applications to view distributed heterogeneous computing resources as a single virtual machine. The toolkit also provides a bag of services from which developers of specific tools or applications, can select to meet their own particular needs. Globus is constructed as a layered architecture in which higher-level services can be developed using the lower level core services [4] and emphasis on the hierarchical integration of Grid components and their services. This feature encourages the usage of one or more lower level services in developing higher-level services.

Higher-level tools such as resource brokers can perform resource discovery by querying its LDAP [60] service. Globus offers QoS in the form of resource reservation and provides scheduling components as part of its toolkit approach. It does not however supply scheduling policies, but relies instead on higher-level schedulers.

The major research challenges and ambitious goals that are addressed in the Globus project are designing, developing and supporting :

- resource management infrastructure.
- fault tolerance support.
- advance reservations and policies for accounting in large-scale grid environments.
- distributed infrastructures and tools for data-intensive applications, managing and providing high performance access to large amounts of data (terabytes or even petabytes).
- new problem solving techniques, programming models, and algorithms for Grid computing.
- high performance communication methods and protocols.

We aim at providing a system for offering access to medical images, which can be interfaced from other external systems. One of these systems can be a Grid (using Globus or other core Middleware), which use it as a service offering access to large amounts of medical data (data-intensiveness) with constraints of high performance. Standards have emerged to structure grid middlewares into a set of services , to integrate new services into an existing middleware and to interface services with external components. Thus, interfaces between a grid using Globus and a service like the one we intend to design should be OGSA [137] or WSRF [163] compliant.

[2]Several tools projects as Condor/G, the European DataGrid project, the EGEE project, have used Globus components. Currently the Globus researchers are working together with the High-Energy Physics and the Climate Modeling community to build a data Grid [6]

OGSA/WSRF

The Grid computing trends are dependent on a common set of standards providing a collaborative context in order for partners to work together. GGF's [162] OGSA Working Group works in this definition.

The Open Grid Services Architecture (OGSA) is an evolution toward a Grid architecture based on services concepts [58] and technologies. It supports [11] the creation, maintenance and application of sets of services maintained by Virtual Organizations (VO).

There are two major pieces in OGSA :

- **Core Grid Components :**
  They target functionalities like resource allocation and policy management. Built on top of the infrastructure piece, these components can be combined to build Grid applications and services.
- **a common Grid infrastructure :**
  the Open Grid Services Infrastructure or OGSI specifies a Grid Service. Such a service removes the need for core Grid components to directly reference specific protocols like FTP, HTTP or LDAP.

Nowadays, OGSA compliance [137] can be defined as clients or services that are compliant with the OGSI Grid Service Specification. An OGSA-compliant service can be defined as any OGSI-compliant service whose interface has been defined (by the GGF OGSA Working Group) to be a standard OGSA service interface.

The OGSI specification defines conventions and extensions based on WDSL and XML schema in order to enable state-ful Web Services. It also [59] : (i) defines mechanisms for creating, naming, and managing the lifetime of instances of services, (ii) manages asynchronous notification of service state changes, (iii) declares and inspects service state data, (iv) handles of service invocation faults, and (v) represents and manages collections of service instances.

The Web Service Resource Framework (WSRF) [163] was proposed (January 2004) as an evolution of OGSI. The WSRF exploits new Web services standards like WS-Addressing. It retains the functional capabilities present in OGSI and changes : (i) some of the syntax, (ii) its terminology, and (iii) its presentation. It also partitions OGSI functionality into five distinct specifications. WSRF can be viewed as a straightforward refactoring of the concepts and interfaces developed in the OGSI V1.0 specification, in a manner that exploits developments in Web services architecture.

At this point, WSRF is proposed as a mean of expressing the relationship between state-ful resources and Web services, and is becoming a substitute for OGSI in OGSA.

Other Middlewares

As Globus, the core middlewares below, must work with data stored in their name-space.

- The **Condor Project** [127] is a High Throughput Computing [61] environment that can manage very large collections of distributed workstations and clusters that are owned by different individuals. It is known for harnessing idle

computers CPU cycles [19] (cycle stealing [3]), but it can be configured also to share resources.

The Condor's architecture is layered and offers resource management services for sequential and parallel applications. It aims at harnessing the capacity of collections of workstations and clusters for applications having a need for huge computer power and heterogeneous distributed resources. It (i) offers resources otherwise wasted by putting idle machines to work, (ii) expands the resources available because it offers machines which are otherwise inaccessible.

Even though Condor itself was not originally aimed for Grid use, it can be considered such as in a computational Grid tool [16]. It has no QoS support and the information store is a network directory that does not use X.500/LDAP [60] technology.

- The Grid Operating System **Legion** [128] [62] [63] is an object-based meta-system or Grid operating system whose goal is to promote the principle design of distributed system software by providing standard object representations for processors, data systems, and hardware and software resources in general. These standard objects represent the base in order to design distributed applications. Its software infrastructure integrates a system of heterogeneous, geographically distributed, high performance machines, and allows it to seamlessly interact. Legion provides application users with a single, coherent, *worldwide virtual computer*.

  Legion works as a middleware layer between the operating system and other Legion resources. It schedules and distributes the programs on available and appropriate hosts; there is not a centralized control of resources, so each resource is independent. This approach of managing resources can be used to parallelize programs. Since all elements in the system are objects, they can communicate to another regardless of location, heterogeneity, or implementation details thereby addressing problems of encapsulation and interoperability. A similar object-oriented middleware research project is Globe [164] at Vrije University, Netherlands.
- **UNICORE** [165] [65] is a Grid computing environment that facilitates secure access to resources in a distributed environment, and eases relocation of computer jobs, for both end users and Grid sites.

  The UNICORE middleware follows a three-tier architecture [64] : (i) user tier, (ii) server tier, and (iii) target System tier.

  The two main areas of UNICORE are : 1) seamless specification of some work to be done at a remote site and 2) transmission of the specification, results and related data.
- **The Gridbus** Project [158] [66] is an open-source, multi-institutional project which aims at designing and developing service-oriented cluster and grid middleware technologies. It uses economic models [16] for efficient management [4] of shared resources and promotes commoditization of their services.

  Gridbus emphasizes the end-to-end QoS for the management of distributed

[3] Condor takes wasted time and puts it to computational use.

[4] The computational concept of *economy* allows defining services providers and offering paid services. They are based on the requirements and budget of particular application and users.

computational, data, and application services.

- The **SRB (Storage Resource Broker)** [166] is a client-server based middleware which provides uniform access to different types of storage devices, resources, and replicated data sets in a heterogeneous computing environment. It supports storage systems such as HPSS, and database objects managed by DB2, and Oracle. It also works together with the Metadata Catalog (MCAT) [167] in order to provide a mechanism for storing and querying system-level and domain-dependent metadata [5] through a uniform interface. Together the SRB and MCAT servers provide a way to access data sets and resources through querying their attributes instead of knowing their physical names or locations. In the SRB environment [68] [67], the physical location of a data set is logically mapped to a logical name of the data sets, hence these may reside in different storage systems. Its arrangement in directory-like structures is called a *collection*, and provides a logical grouping mechanism for containing groups of physically distributed data sets *(sub-collections)*.

Grid Middlewares are important in the sense that a *Medical Application* can be concerned to interface Grids developed on top of them.

Such an application must be able to interface different Grids, based on different middlewares, such as Globus, Condor, Unicore, Gridbus, SRB. In this work we do focus on the implementation of such interfaces, but deal with a conceptual model which will enable such an adaptability.

3.1.2 Grids Related Projects

In section 2.5.1 we introduced ongoing work in the medical application field, thus, we described grid related projects such as Mammogrid, e-DIAMOND and others. In this section we describe multi-application grid projects which also address middleware topics. The target applications of these projects are more general than the specific medical field.

DataGrid/EGEE

The Data Grid project [126] was born as an initiative of the High-Energy Physics (HEP) community, CERN and the European Organization for Nuclear Research, in order to provide intensive computation and analysis of shared large-scale databases (several Petabytes) across widely distributed scientific communities like *HEP, Earth Observation, Biological and Medical [21] informatics*, all communities that are facing equally daunting challenges to cope with huge floods of data. The project has finished (February 2004) but it has evolved into another bigger project, EGEE [143], which aims to implement a production environment at the European level.

The project objectives [19] were : (i) to establish a research network for data Grid technology development, (ii) to demonstrate data Grid effectiveness through the large-scale real world deployment of end-to-end application experiments, and

[5] Metadata associated with data sets, users and resources, information for access control, and data structures

(iii) to demonstrate the ability to use low-cost commodity components to build, connect, and manage large general-purpose, data intensive computer clusters.

The project focused on developing middleware services able to be used in the analysis of physics data. It based on Globus (with extensions for data Grids) and looked at being distributed in a hierarchical fashion into multiple sites worldwide. It has dealt with global name-spaces in order to handle the creation, access, distribution, and replication of data items. Special workload distribution facilities were implemented for balancing the analysis of jobs in the Grid, and maximizing the throughput from several hundred physicists. Application and user access monitoring were also used to optimize data distribution.

The Data Grid project was implemented over a hierarchical machine organization with less data stored at lower levels of the hierarchy. CERN was the *Tier 0*, storing almost all relevant data with several *Tier 1* regional centers in Italy, France, the UK, the USA, and Japan, all of them supporting smaller amounts of data. An extensible schema based on a resource model with a hierarchical name-space [20] organization was implemented. The Data Grid did not offer any *QoS* and the resource information store was based on an LDAP [160] network directory. Resource dissemination was batched and periodically pushed to other parts of the Grid, and resource discovery in the Data Grid was decentralized and query based.

The *LHC (Large Hadron Collider)* [168] Computing grid project *(LCG)* [169] has collaborated directly with DataGrid's people, and has adapted a version of the *SRB Resource Broker* [166] to be used as a Grid middleware component. This middleware has been also highly influenced by the Grid system developed by *ALICE* [6], called *AliEn*. The core infrastructure of LCG *(LCG-1)* has been improved to *LCG-2* and taken as the base core middleware for the new *EGEE project* [143], which aims at developing a service grid infrastructure (in Europe) available to scientists 24 hours-a-day [7].

The work described in this thesis must be interfaced with middleware like LCG2 in order to offer services of medical data access and execution of hybrid queries over medical images. However, this is an integration at the application level of the Grid, and not at the middleware layer.

Other Grid Projects

There exist so many Grid Projects [159] around the world, that we cannot mention all of them; thus we selected the most significant. The projects below aim at developing core middlewares or in using existing ones in order to develop computing [106] [26] or data grids.

- The EuroGrid [172] project was granted by the European Commission and was a successful initiative for experimenting with the use of GRIDs in selected scientific and industrial communities. The project consisted of application-specific groups for (i) transparently enabling chemists and biologists to submit

[6] LHC at CERN has four (4) experiments in High-Energy Physics (HEP) : LHCb, CMS, ATLAS and ALICE.

[7] The EGEE project is among the largest of its kind, with a budget of over 30 Million Euros for 2 years. In the year 2006 it could be extended for another 2 years with an extra-budget, and after 2008 it could become the ARDA Grid [170]

their work to HPC facilities *(BioGrid)*, (ii) advancing toward a weather prediction portal *(Meteo GRID)*, (iii) experimenting with CAE (Computer-aided Engineering) applications, which are huge computing power consuming applications *(CAE GRID track)*, and (iv) researching in HPC computing *(HPC Research GRID)*.

The EuroGrid was based on the UNICORE [165] middleware.

- K*Grid project [171] is an initiative in Grid researches supported by the MIC (Ministry of Information and Communication, Republic of Korea), which aims at providing a research environment to both industries and academia, with access to a huge amount of computing power and virtual experiment facilities. A key goal is to develop a Grid middleware which integrates many geographically and organizationally dispersed computing resources, massive data, and human power. It uses Globus for developing the components in the application middleware.
- The NASA's Information Power Grid (IPG) Project is a high-performance computing and data grid which allows scientists and engineers throughout NASA [175] to access widely distributed heterogeneous resources from any location with the IPG middleware.

  NASA looks for revolutionizing its computing processes by implementing fundamental changes and improvements in access to powerful computing systems, large-scale data archives, scientific instruments, and collaboration tools. Its vision aims at offering services which are highly capable of providing transparent access to these resources, regardless of their location or exact nature [8].
- The CrossGrid [130] main objective is to extend the GRID environment across Europe, and to a new category of applications. It implements and exploits new Grid components for interactive computation and data intensive applications, and addresses realistic problems in medicine [9], environmental protection, flood prediction, and physics analysis [130] [71].

  The applications focused on the project are characterized by the interaction with a person in a processing loop, which means that responses to an action by an user are required from the computer system. These responses can happen from real, through intermediate, to long time, and they are simultaneously computed.

  The CrossGrid project develops applications such as : (i) pre-treatment planning in vascular interventional and surgical procedures through real-time interactive simulation of vascular structure and flow, (ii) support System for flood prevention and protection, (iii) High-Energy Physics (HEP) applications for physics analysis running in distributed mode, and (iv) weather forecast and air pollution modeling.

  The distributed and interactive nature of these applications motivates the use of a Grid-specific software architecture. It enables technologies and services by

[8] The IPG project participates in the Grid Forum [162] and has developed its middleware using Globus [129] and Condor [127] job management system for workstation cycle scavenging. It uses also SDSC's Metadata Catalogue (MCAT) [167] and the Storage Resource Broker (SRB) [166]

[9] e.g., simulation and visualization of surgical procedures ; in this way, medical doctors would use new tools to help them to obtain correct diagnoses and to guide them during operations.

developing middleware and toolkits, which include resource scheduling, performance prediction, monitoring, parallelization, user access via portals, user-friendliness, ubiquitous computing and access to information [10].

- The Datacentric Grid (DCGrid) [174] is an innovative concept [69] [70] of Grid, proposed by David Skillicorn at Queen's University in Canada, which aims at designing and implementing grids for data-intensive operations in which data are moved as little as possible. Datacentric grid applications are likely to require both access to data, and large amounts of computation, and aim at reversing the traditional view that processors are the critical resource in systems and hence that data should move to processors. This model aims at moving the software to the data instead of moving data to the software, as the classic model does. Hence, moving code requires orders of magnitude less bandwidth than moving the data; this means deeply changing *processor-centric* computing technology into a *datacentric* one.
  This change of approach requires a wide-ranging change in the way of approaching distributed computation, so it concerns problems in fundamental informatics such as : (i) mobile code, (ii) distributed data processing, (iii) fragmentation code, (iv) security, privacy, and confidentiality, and (v) mobile users.
  We did not work with this approach, thought it has some interesting properties to medical image applications. Indeed, because of the restriction of access to data, it would be better to go to the data rather than copying the data. In this way, confidentiality could be kept, and access restriction could be assured. However, this approach brings constraints of computing power, because the site having the data must also have the computing resource. This isa big issue for a medical site : medical institutions are looking for processing power and not for offering it.

3.2 Distributed Computing Technologies

3.2.1 Peer to Peer Computing

Peer-to-Peer is defined as a class of applications that take advantage of resources (storage, cycles, content, human presence) and also claims to address the problem of organizing large scale computational societies, as grid computing does. It accesses decentralized resources and operates in an environment of unstable connectivity and unpredictable IP addresses. It also has independence from DNS and is autonomous from central servers.

Intel P2P working group defines P2P as "The sharing of computer resources and services by direct exchange between systems" [72]. P2P systems have such characteristics [73] as : (i) scalability, and (ii) reliability.

P2P can be categorized into two groups classified by the type of model : *pure P2P, and hybrid P2P*. The pure P2P model does not have a central server. Hybrid P2P models employ a central server to obtain meta-information such as the identity

[10] The CrossGrid software is based on the Globus Toolkit [129] and the EU DataGrid components.

of the peer on which the information is stored or to verify security credentials. In a hybrid model, peers always contact a central server before they directly contact other peers.

By topology, P2P systems are classified [74] as : (i) centralized, (ii) decentralized, (iii) hierarchical, and (iv) ring systems. There is always a common feature : file transfers and control messages are always done directly between the peer offering a service (*e.g.*, file sharing) and the peer requesting it [75].

In a **centralized topology** the client contacts the server to inform it of its current IP address and names of all the files that it is willing to share. **The ring** is made up of a cluster of machines that are arranged in the form of a ring to act as a distributed server. In a **hierarchical topology** authority flows from the root name servers to the servers connected to the root and so on. In the **decentralized** [75] topology, all peers are equal, hence creating a flat, unstructured network topology. However, what really happens are mixtures of them, hence creating hybrid topologies.

A fundamental problem in P2P applications is how to efficiently locate the node that stores a particular data item. Therefore, some scalable indexing mechanisms called *Content -Addressable Networks (CAN)* have been defined. CANs resemble a distributed hash table of *(key, value) pairs*, which efficiently map *"keys"* onto *"identifiers"*. Given a key, the CAN allows one to map the key onto a node. Data location can be easily implemented by associating a key with each data item. Then, the key data item pair can be stored in the node to which the key maps.

As a Content-Addressable Network, Chord [140] [76] [2] [3] [77] and Freenet [176] are outstanding proposals. In terms of Metadata distribution, interesting technologies and ideas are offered by Mojo Nation [179] and JXTA Search [161] [57].

The *Napster* Project [177] uses the centralized topology [78], and *Gnutella* [178] the decentralized one [78]. A good example of hybrid architecture [79] [80] is *OpenFT* [180] . Other interesting P2P projects are,

- **Oceanstore** [181], a project which aims at providing a global-scale persistent data store.
- **FastTrack** [182], is also a hybrid P2P architecture which is used by Kazaa.
- **Kazaa** [183], is a file sharing P2P system.

Our goal is to learn from these technologies and use them in the definition of the distribution layer in our architectural proposal (next chapter). As we saw in chapter 2 medical data are geographically distributed and problems of access arise due to confidentiality. A medical system can manage data by regions but communication between regions can take advantage of a P2P approach. data by regions, whereas communication between regions can use a P2P approach.

.

3.2.2 P2P vs Grid

Both environments are concerned with sharing resources within Virtual Organizations (VO) ; however there are some important differences [18] between them :

- Grids provide services to moderate-sized communities and offer integration of resources which allow one to deliver nontrivial QoS. In contrast, P2P deals with

thousands (or millions) of participants, but offers specialized services which are less concerned with QoS.
- The P2P communities are composed of anonymous individuals with little incentive to act cooperatively. The Grid communities are established and have VOs where users must be known and can be controlled, their engagement is often-limited, and a membership exists.
- Grid systems integrate resources (cluster, storage system, database or scientific instrument) that are more powerful, diverse, and better connected than resources within a P2P network.
- Resource availability is higher and more uniform in a Grid environment than in a P2P one.
- Grid applications are far more data intensive than P2P applications.
- Grid efforts have gone on to define a complex service-oriented architecture within all services have standard interfaces and behaviors. P2P systems have tended to integrate simple resources (individual PCs).
- Functionality requirements can be different, *e.g.*, Grids might require accountability and the P2P system anonymity.

3.2.3 CORBA, DCOM and Java/RMI

The *Common Object Request Broker Architecture (CORBA)* [134] is an open distributed object computing infrastructure being standardized by the Object Management Group (OMG) [191]. CORBA specifies a system which provides interoperability [104] [102] between objects in a heterogeneous, distributed environment and in a way transparent [105] to the programmer. Its design is based on OMG Object Model, which defines semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way ; clients request services from objects through a well-defined interface, by issuing a request (event) to the object.

The central component of CORBA is the Object Request Broker (ORB). It encompasses all of the communication infrastructure necessary to identify and locate objects, handle connection management and deliver data. Its basic functionality consists of passing the requests from clients to the object instances on which they are invoked.

The *Distributed Component Object Model (DCOM)* is an extension of the Component Object Model (COM) [192] [100] that allows COM components to communicate across network boundaries. Traditional COM [11] components can only perform interprocess communication across process boundaries on the same machine.

The third most popular distributed object paradigm is *Java Remote Method Invocation (RMI)* [193], which relies heavily on Java Object Serialization. Each Java/RMI Server object defines an interface which can be used to access the server object outside of the current Java Virtual Machine(JVM) on another machines, *e.g.,* an object running in one Java Virtual Machine (VM) is allowed to invoke methods on an object running in another Java VM.

[11]The COM (Component Object Model) is a Microsoft technology for Windows-family of Operating Systems

While CORBA and DCOM are standards for defining interoperability, Java RMI is only a set of protocols (developed by Sun's JavaSoft division) that enables Java objects to communicate remotely with other Java objects.

Object Management Architecture (OMA)

The OMA [191] embodies the OMG's vision for the component software environment, and guides standardization application to plug-and-play component software environment based on object technology. An expanded vision of the OMA can be found within the OMG's Model Driven Architecture (MDA).

The OMA Reference Model identifies and characterizes the components, interfaces, and protocols that compose the OMA, while the Object Request Broker (ORB) component enables clients and objects to communicate in a distributed environment : it provides an infrastructure allowing objects to communicate independent of the specific platforms and techniques.

The CORBA Services component standardizes the life cycle management of objects and allows creation of objects, control access to objects and tracking of relocated objects.

The Application Objects performs specific tasks for users. An application is built from basic object classes, and new classes of application objects can be built by using existing classes [124] [125] (inheritance).

These paradigms define interoperability between objects, while we are looking for a highest-level vision of a complete distributed environment. Architectures such as MDA/OMA [191] propose a high level overview of distributed systems (DS) and focus on the global interfaces and integration layers.

3.2.4 Distributed Storage

File Systems

Multiple classes of file systems extend beyond the bounds of a single computer; they include network file systems and distributed file systems (client/server file systems built around a network data access protocol), and also include parallel and cluster file systems where many computers act as peers.

The *Sun's Network File System (NFS)* [83] is a single network protocol for taking local file system requests and redirecting them between computers. It does not include an actual file system managing data on disks, but relies on local file systems at the servers to store data. Its prevalence has made it a platform for further development.

Microsoft's CIFS file system [84] defines a session-oriented data access protocol with which clients connect to and mount a remote name tree into the local file system. These network file systems are limited in that clients must forward all requests to a server. Many factors limit performance in this environment, including network data transfer rates, end-to-end latency, and the bottleneck of servers. Additionally, servers cannot act in concert and therefore fail to provide load balancing, parallel access, takeover on failure, or a uniform global name space for files and directories.

The Network Appliances Write-Anywhere File Layout *(WAFL)* [85] is a customized local file system designed for serving network file system requests. It adds transactional capabilities to reduce latency and increase data throughput, and has an efficient file system checkpoint mechanism to make data highly available and speed the recovery process.

The *Andrew File System* (AFS) [86] and its successor the *Distributed File System (DFS)*[12] [87] expand the client/server file system concept to distributed file systems by enhancing security and scalability, as well as providing a uniform global name space for all clients. Distributed file systems provide a uniform global name space, so that files appear with the same name on all clients, and all portions of the name space can be accessed using file system methods. Servers cooperate to provide takeover and load balancing features and management tasks, such as coordinated backup and recovery. However, existing distributed file systems require servers to access file data on behalf of clients and ship that data to the client.

NFS and CIFS are only distributed data access protocols, while AFS/DFS is a distributed file system.

There also exists research in other kinds of file systems such as the *Parallel file systems* [13] which offer a high performance alternative to distributed file systems. However, they rely on the high-speed communication facilities of supercomputers and do not translate well to weakly connected environments without high-speed networking hardware (*e.g.*, a medical site). *Hierarchical File Systems* [14] achieve scalability and performance by building a file system over a distributed logical volume service; it makes many computers appear to share the same storage subsystem, shielding the file system from many issues in distributed data management. *Serverless File Systems* [15] achieve scalability and performance for distributed file systems by removing the bottleneck of a centralized file server. Finally, *Replicated File Systems* [16] replicate data and state among their many computers, to make data highly available and fault tolerant. They are constructed on top of a group services middleware that performs the replication, failure detection, and consistency protocols.

Such *Distributed File Systems* offer pertinent functionalities for managing distributed medical data : performance, scalability, fault tolerance. However they require a strongly unified system administration which is clearly incompatible with the reality of the highly decentralized health organization in our countries.

Hierarchical Storage Management (HSM)

The massive explosion in data volumes represents one of the hot areas of computing and currently one providing severe restrictions for computational growth. Problems such as : (i) increasing computational performance, (ii) easier and higher bandwidth access to data from the Internet, (iii) sophisticated increases in algorithms, and (iv) addressing the issues regarding legacy storage systems; are all

[12] There also exist CODA [201], which is a networked file system also based in the AFS
[13] *e.g.* Tiger Shark, PPFS, and SPIFFI
[14] *e.g.* Frangipani, Cambridge
[15] *e.g.* xFS, JetFile
[16] *e.g.* Calypso, HARP, high availability NFS

contributory factors leading to the requirement to provide the capability to offer data repositories with high performance and high availability.

A response to this problematic is the Mass Storage Systems *(MSS)*, and specially its sub-type of *Hierarchical Storage Management (HSM)* systems.

A *(HSM)* consists [22] of several layers of storage media :

- *a fast on-line primary storage, e.g.,* RAID disk system (10s to 1000s GB storage).
- *off-line or automated secondary storage*, is a slower media, possibly slower disk, tape cartridges or cassettes, which could have automated access via robotics (until some petabytes of storage per robot silo).
- *off-site tertiary* storage for archiving.

some characteristics of an HSM system are : (i) the data movement between software layers is transparent and automatically handled by the system, (ii) the storage capacity increases by layers, whereas, inversely, the cost and the performance decreases, (iii) files are automatically recalled from offline when they are accessed, (iv) all data (petabytes of data and billions of files of varying sizes) appears to be on-line, (v) there is no need for a separate backup of the files, and only the most recently referenced files are kept on disks, (vi) from the user's point of view, the files are always visible, and (vii) it is a cost efficient way to store huge amounts of data.

The most accepted standard for this kind of storage is the IEEE Mass Storage System Reference Model (MSSRM), version 5 [50], and it is used specially for HPSS and Storage Tank.

The most significant HSM for High-Energy Physics (HEP) community, at the moment are :

- *Castor* [132], the HSM system at CERN. It has good scalability, high modularity to ease replacement of components, and availability to work in UNIX and windows-NT systems.

  The Castor user's client has two components : (i) the *stager* to trigger the migration and recall of data, and (ii) *RFIO (Remote File I/O)* to access the data on a remote disk pool. It has been stress-tested at CERN by sending 100TB of data at an average rate of 85 MB/s. Its average tape moving traffic is about 30 TB per week for the Alice Data Challenge experiment.
- *Enstore* [185] [82], a tape robot management program developed to handle large volumes of data at Fermilab (USA). It was designed to give users access to tape data as conveniently as to files on their native file system.

  The system scales very well in I/O, typically a separate *mover* [17] handles each tape drive and transfers are delegated to a specific *mover*. So far, up to 10 TB of daily data movement has been demonstrated. More general access is provided through a disk cache. This is an area of ongoing work in a collaborative project *(dCache)* [186] between Fermilab and DESY [194] laboratories.
- *EuroStore* [184], which addresses the problem of handling the large volumes (several PBs per year) of data. It consists of two major parts : the parallel file system (PFS) and a HSM developed by DESY. The HSM itself, which is based on the IEEE mass storage model, interacts with the robotics and any physical storage devices. It was developed for dealing with data expected from

[17] A mover is a software component in charge of moving file between storage layers

LHC experiments at CERN.

- *HPSS (High Performance Storage System)* [133] [81], which is a HSM system that provides services for very large storage environments. Its network-centred design (based in the IEEE Mass Storage Reference Model), allows data to be moved from an intelligent disk or tape controller to the client. HPSS Movers are claimed to deliver data at the full device speeds (about 50 MB/s) single stream. The system allows for parallel data streams from multiple storage devices. It is claimed to give aggregate throughput rates of about 1 GB/s.
  Data transfer takes place directly between the client and the storage device controller. HPSS was designed for the massive parallel processors where large data files and high transfer rates (ca 100 MB/s) are often called for. The system is highly scalable.
  With HPSS, files can be (i) stored, retrieved, copied, moved, replaced or deleted, (ii) organized in simple or complex tree structures, and (iii) shared with selected individuals or an entire user community. Once files are transferred to HPSS, they are stored in an archival system. HPSS transparently manages the storage hierarchies, migrating, purging and caching files as required based upon the dynamically changing environment.

HSM are specially used in data-intensive scientific experiments like physics of particles, meteorology and astrophysics, but due to the increasing volume of data, it is becoming a necessary tool for biology, genetics and medical imagery. However, this is a very expensive technology for hospitals and medical centers. A medical system does not offer storage services as a HSM does, but its distributed structure, and its possibility of accessing multiple data repositories, can enable the cooperation with existing HSM in a direct way or through a Storage Element of a Data Grid. In this way, the system we intend to develop, by moving data between medical sites, servers, and a grid or HSM, *de facto* offers a service of data storage.

3.3 Images Storage

3.3.1 DICOM3

The Digital Image and COmmunication in Medicine **DICOM** [142] [18] specification has emerged as the standard for medical image storage [49]. It facilitates interoperability of medical imaging equipment by specifying protocols for medical devices, syntax and semantics of commands and associated data for transmission information ; it facilitates operations in a networked environment [49].

DICOM describes an image format, a communication protocol between an image server and its clients, and other image related capabilities. On top of such a standard, Picture Archiving and Communication Systems *(PACS)* can be deployed for allowing managing of data storage and data flow inside hospitals.

Some popular software systems which implement the DICOM3 standard *(DICOM toolkits)* are **CTN and DCMTK**. The DICOM Central Test Node Software *(CTN)* [135] is a DICOM implementation which was designed to be used at the

[18] DICOM is an evolution of an old standard called ACR-NEMA

RSNA (Radiological Society of North America) annual meetings to foster cooperative demonstrations by the medical imaging vendors. The goal was to provide a centralized implementation to test vendor's products on a common plate following the DICOM standard. CTN is a medical data and images storage system (using DICOM 3 format), but is not a Mass Storage System (MSS). *DCMTK* (DICOM Toolkit) [136] is a collection of libraries and applications implementing large parts of the DICOM standard for medical image communication. It includes software for examining, constructing and converting DICOM image files, handling offline media, sending and receiving images over a network connection, as well as demonstration image storage and worklist servers. While CTN is stronger in its server implementation, DCMTK offers a better development toolkit. There are others toolkits such as Osiris, Java DICOM Toolkit, DICOM DLL, AN/API DICOM Toolkit and DICOM Suite ; due to the wide acceptation of the DICOM standard we will use it as reference for our applications.

3.3.2 PACS and RIS

PACS (Picture Archiving and Communications System) are systems composed of various components for transmitting, storing, displaying and archiving patient data. Usually PACS distributes images and its associated reports throughout a medical system, and integrates it to the Hospital Information System (HIS) or Radiological Information Systems (RIS).

A PACS should allow the functional ability not only to distribute images to the requestor but also communicate to physicians for patient scheduling, to download patient demographics to modalities, to track image location, and to assemble collaborative material necessary to interpret the image.

The PACS archives the images and allows image transfers. The HIS/RIS contains full medical records : image-related metadata and additional information on the patient history, pathology follow-up, etc. Although some vendors propose integrated PACS and HIS/RIS, there exists no open standards for the data structure and the communication between the services in this architecture. Moreover, they are usually designed to handle information inside an hospital but there is no system taking into account larger data sets nor the integration with an external component such as a computation/storage grid.

PACS makes use of backup and archival systems, and this is where a link to Mass Storage Systems (MSS) comes in. A PACS is not a MSS, but the same data-intensive constraints apply in using a PACS system, as that of a MSS one.

There exist many PACS systems installed in the world. We can make reference to The Geneva PACS, The Hammersmith Hospital PACS or The Sheba Hospital PACS in Tel-Aviv.

In this thesis, we will assume that image data within a hospital are stored in a DICOM compliant Server or PACS so that we can make use of DICOM protocols to retrieve raw image data and the associated metadata.

Chapitre 4

Distributed System Engines (DSE Architecture)

*"... the key concept is the ability to negotiate resource-sharing arrangements among a set of participating parties (providers and consumers) and then to use the resulting resource pool for some purpose.", **Ian Foster**, Argonne National Laboratory, USA*

–

Summary 4

We define an architecture which addresses the problems of Distributed Computing and Medical Image Processing, and we then describe its usefulness for building systems.

The architecture (Distributed System Engines (DSE)) proposes a way of building distributed systems (DS) in an environment with strong requirements for high performance, and with characteristics of extensibility, scalability and openness. These DS are composed of engines. The architecture has a pyramidal definition through five layers which increase in semantic significance : (i) DSE^0, the lowest level, defines a message passing interface in charge of the transmission of messages between multiprocess programs, (ii) DSE^1, defines a transaction structure built over the message passing level. (iii) DSE^2, offers distributed facilities, (iv) DSE^3 is the application layer, and also offers a programming interface (API) so that a user application can be built on top of the underlying distributed system, and (iv) DSE^4, the user layer, offers high level access to data, metadata and services.

Furthermore, the ***DSE architecture*** *has an horizontal definition by each one of the layers, and is based on a multi-process structure which enables the exchange of messages between processes. In this way, it is possible to define entities of a higher level of semantic significance, called* ***Drivers****, which deal with different kinds of* ***transactions : queries, tasks and requests****. We define also a different kind of driver per each kind of defined transaction. In a higher level, an aggregation of drivers allow us to define* ***services****. This architectural framework of drivers and services eases the design of components of a Distributed System (DS).*

*There exist two big groups of layers : middleware (first three ones) and application (last two ones). The middleware layers are implemented in a prototype (****DSEM****), and then used for developing a medical application (****Distributed Medical Data Manager (DM^2)****).*

–

4.1 Our Project

A classical definition of a Distributed System (DS) was given by Mullender [1] as "several computers doing something together". Thus, a Distributed System (DS) contains multiple computers, interconnections, and a shared state which comes from the computers cooperation [1].

Our view of a DS is a set of intercommunicating and cooperating virtual components which we divide between **engines** and **external tools and services** (also called machines). Each engine, by itself, is a complex component composed by a set of independent local processes [1], which interact by exchanging messages, and connected to the external world (tools, services and other engines).

We propose an architecture **(Distributed Systems Engines - DSE)** that deals with designing and implementing these kinds of engines in an environment with strong requirements of high performance. Our architecture modelize not only the DS, but also the internal structure of each one of its components (engines). It is based on the definition of four different kinds of **drivers** which manage three types of entities : messages, transactions and tasks. This decomposition allows us to define highly scalable and extensible engines, which are easily adaptable to the design of systems dealing with medical hybrid queries.

We have also implemented a prototype of the architecture (**DSEM**, or **DSE the manager**). It is composed of a set of libraries, APIs, tools and drivers (middleware style), which allow us to develop systems dedicated to medical image processing, and especially to the resolution of queries by content. These systems aim at managing huge volumes of images and metadata in a distributed environment and interface computing and storing Grids.

The architecture (DSE) is analyzed in this chapter, its prototype implementation in chapter 5, the prototypes applications and the performance concerns in chapter 6.

4.2 Pyramidal architecture

We understand a *Distributed System (DS)* as a set of *distributed system engines (DSE)* [2] interfacing external machines :

DS $\leftrightarrow$ {DSE} U {machines} [3]

The *machines* are defined as external software elements to a DS. They represent tools and services, and there is not possible to modify them : they exist as they are, and all we can do is to interface them. Usually, they represent tools and services

[1] In this document we will not make any difference between processes and threads; we consider that question as an implementation problem which does not strictly concern the architecture definition

[2] For simplicity, in this text we refer to engines instead of distributed system engines.

[3] A distributed system is the union of a set of engines and a set of machines (external tools and services).

which the engine can use, *e.g.* a cache utility, a database service, a grid computing service, etc. The *Engines* are complex components, composed of interacting local processes which exchange messages between them, and which also have interactions with external machines and additional engines. An *Engine* is the *basic unit of a Distributed System*, or a brick to build the D.S., but in its internals, each engine is a set of interacting entities which we define below as *Drivers*.

The engines are the objects that we develop for building the infrastructure of one DS. The machines are the external components to interact with (tools and services).

DSE is a multilayer architecture specially designed for building engines able to manage distributed systems. The proposed architecture goes through five semantic layers, from the message layer to the application layer.

The different layers have an increasing level of semantic significance and allow building systems (at the high layer) which take advantage of the lowest layers. This architecture aims at providing transparency, scalability, extensibility, and enabling high performance implementations.

The DSE architecture uses the message passing paradigm - in its lowest layer - to interconnect processes, and takes advantage of the classical concept of transaction in order to increase the semantic significance of interacting messages. It integrates existing systems and takes advantage of these services.

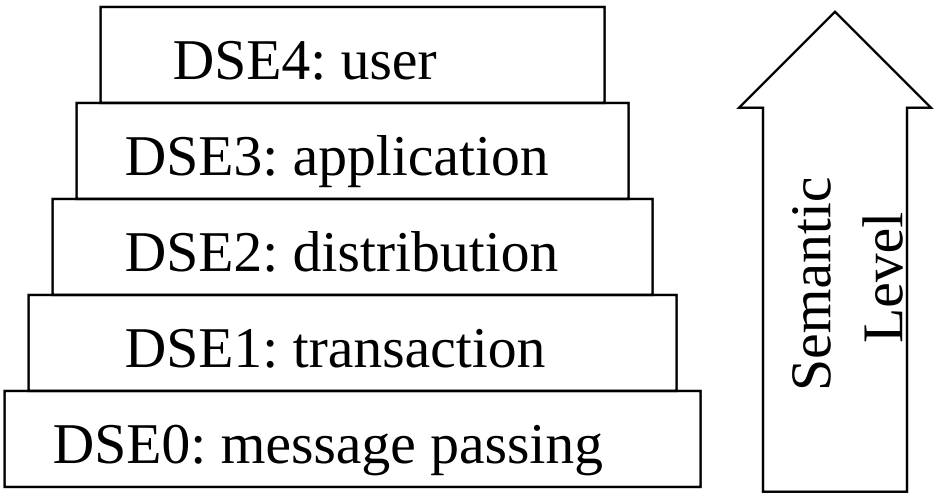


FIG. 4.1 – DSE layers

The architecture has a *vertical (pyramidal) definition* (figure 4.1) through five layers :

- DSE^0 : The lowest level, defines a message passing interface in charge of the transmission of *messages* between multiprocess programs. Its goal is to reach the target process in an efficient way and as quickly as possible.
- DSE^1 : This level brings atomic operations (*transactions*) to process complex requests composed of several messages. Virtually all methods for specifying and reasoning about concurrent and distributed systems are based on a model of

computation that incorporates some notion of atomic actions [1] . DSE[1] defines a *transaction* structure built over the message passing level, which offers the ACID properties : Atomicity, Consistency, Isolation and Durability [1].
- DSE[2] : Takes advantage of the two lowest levels in order to offer *distributed* facilities for dealing with distribution over several engines.
- DSE[3] : Is the *application* layer. An application integrates communicating components build on the lowest layers.
- DSE[4] : Is the *user* layer. It allows one to define external interfaces.

The first three layers deal with *middleware functions*, and the last two are *application* oriented. In this document we will refer to *middleware layers* or *application layers* in order to group the first three layers on the last two.

The detail of each layer corresponds to an *horizontal definition* of the architecture, and is discussed in sections 4.3, 4.4, 4.5, 4.6 and 4.7.

4.3 DSE[0] : Message Passing Engine Layer

4.3.1 Message Passing Technology

Message Passing (MP) is a programing paradigm in which the user directly controls the flow of operations and data within his/her parallel programs. A message-passing library allows the programmer to explicitly order each processor what to do and provides a mechanism for transferring data between processes. It is widely used in the field of parallel computing due to the following advantages :
- Hardware match of separate processors connected by a communication network.
- Functionality for expressing parallel algorithms.
- Achievement of **performance** by giving the programmer an explicit control on the data locality and transfer.

Its main drawback is the responsibility it places on the programmer. The programmer *must explicitly implement a data distribution scheme with all interprocess communications and synchronizations*. In doing so, it is the programmer's responsibility to solve data dependencies and avoid deadlocks and race conditions. At present, middleware layers that compose Grids and huge Mass Storage Systems, are built as multiprocess systems which communicate between them by issuing messages.

A common use of message passing, is for communication in a parallel supercomputer. A process running on one processor may send a message to a process running on the same processor or another. The model also fits well on clusters of workstations which are composed of separate processors connected by a communications network, or within computers running many subtask processes where shared memory mechanisms can be used in order to implement MP.

This paradigm allows an application to be structurally separated in different processes and to have communication by issuing messages between them [36] . At the moment the most popular libraries are PVM (the Parallel Virtual Machine [36]) and MPI (Message Passing Interface [43]). There also exist communication libraries developed in France, as PM[2] [113] and Madeleine II [114].

An engine can be developed by using the message passing tool that the developer wants, however, our implementation (see section 5.2.1 in the chapter 5) uses a proprietary library which will be presented in the next chapter. Similarly, we propose a daemon for dealing with the routing of messages, called MPK (Message Passing Kernel) ; the conceptual definition is described below (section 4.3.2), and in the next chapter (section 5.2.2) a prototype implementation is also showed.

PVM

PVM [141], written at Oak Ridge National Laboratory in 1989, is a portable heterogeneous message-passing system. It provides tools for interprocess communication, process spawning, and execution on multiple architectures. The PVM standard is well defined, and PVM has been a standard tool for parallel computing for several years.

PVM is built around the concept of a Virtual Machine which is a dynamic collection of computational resources managed as a single parallel computer, and provides heterogeneity, portability and encapsulation of functions [35].

MPI

MPI [39] [43] [139] has come into the mainstream more recently than PVM, but it is a mature standard that has been available for several years. The most popular implementations were written at Argonne National Lab (MPICH) and Ohio Supercomputing Center (LAM/MPI). MPI is intended primarily for data-parallel problems [17]. Therefore, it does not have the flexibility of PVM dynamic process spawning, but its collective operations (like gather-scatter operations) and asynchronous message passing capabilities (asynchronous sends and receives) are much more sophisticated [37] and configurable than those in PVM. MPI was developed to serve as a common standard, bringing together years of research and experience with message passing.

There exists a new specification (MPI-2) which extends the current one (MPI-1.2) ; it includes the definition of interfaces in key areas not covered by the MPI-1.2 specification, such as dynamic process management and one-sided communications.

MPI is expected to be faster than PVM within large multiprocessors [38] and has many more point-to-point and collective communication options than PVM. MPI is based only in Message Passing features and does not implement the concept of Virtual Machine ; however it has collective communication routines for communication among groups of processes, and the ability for specifying communication topologies [35].

4.3.2 Layer 0 : Definition and Structure.

At this layer a *Distributed System Engine (DSE)* is defined as the *union* of a set *local Processes* and a *Message Passing Kernel (MPK)* :

DSE^0 $\leftrightarrow$ {processes} U MPK [4]

In layer 0 processes collaborate by exchanging messages through an intermediary which is called the message passing kernel (MPK). These processes only exist in a single host, and therefore an IPC mechanism is sufficient for interaction.

The Message Passing Kernel (MPK).

The message passing kernel (MPK), is an entity in charge of providing routing of messages between *local processes*. The communication between the processes of an engine and the MPK is implemented by IPC (Inter Process Communication [88] [89]) mechanisms.

The main thing that a MPK does is to route messages from process to process. All the processes in between have different types and functionalities. They allow the user to build high level functions and to get access to applications which reside outside the system, that is, in the local machine, in the local network (LAN) or in the WAN.

The MPK uses IPC mechanisms due to the fact of all the processes using it are in the same single host. In order to deal with the internal routing of messages, the IPC is very fast because it is based on memory accesses, and it is sufficient because at this level no network communication is required.

In order to improve performance the MPK must :

- process messages in parallel
- minimize the queue of messages
- have knowledge of *different instances* of target processes in the engine. This allows the MPK to select the best instance for sending a message ; *e.g, a driver* as the onesdefined below in section 4.4, can be implemented as a multiprocess entity, so when a message is sent from a process to this *driver*, the MPK can redirect it to the best process in the *driver*, which means, the one which is less busy.

The MPK does not send messages to the network, it only routes messages between local processes. These processes, as defined below, can issue messages to the network instead of the MPK.

Process types

The system assumes that there are processes which receive messages from the network and other ones which issue messages to the network. The different kinds of processes exchange messages between them in order to process a query (represented as an input message) which has arrived to the engine. Functions as reading or writing to the network, or to an IPC mechanism, are distributed in different kinds of processes.

[4] The formulas that we use in the definition of the different DSE layers do not have a mathematical semantic, these are only used to allow the reader having quickly a compacted overview of the layer.

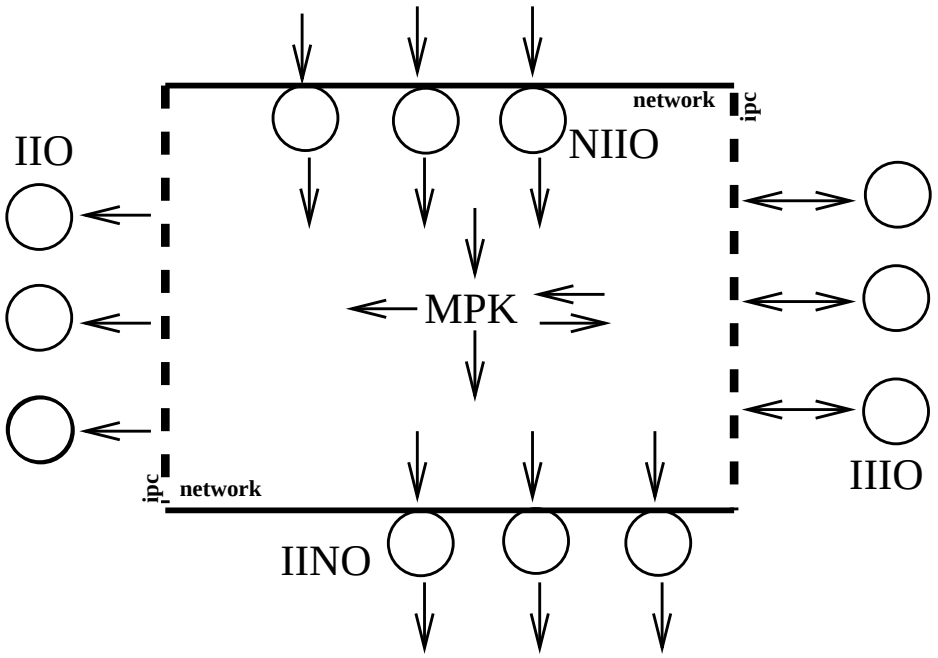


FIG. 4.2 – Message Passing Kernel and Processes Types
The figure shows the network side at top and bottom. The local side (IPC) is shown in sides left and right. Processes are represented as circles. Inside the square is the MPK domain.

The figure 4.2 represents an engine in its lowest layer, the one where only an exchange between local processes happen. Circles represent local processes, *e.g.*, running in the same computer. The square outbounds the engine and its interfaces. Its interior and the vertical dotted lines represent the IPC mechanism, like share memory. The horizontal lines at top and bottom, represent the interface with the network. The arrows represent the direction of the messages. As shown in figure 4.3, a process on top of the figure receives a message from the network, and then delivers additional local messages to the other processes of the engine (in both right and left sides of the figure, or on the bottom side for reaching the network again).

The defined types of processes are :

- Network_in/ IPC_out processes (NIIO). In the top of the figure 4.2 are those processes which have the ability to receive inner messages from the network. These processes are the entry points to the system, and although they also send response messages, the arrows in the figure 4.2 show only its main characteristic : the receipt of requests. In the other direction, these processes do not receive requests from the system itself, their IPC communication work principally in the output direction, issuing requests to another process. The IPC communication is made through the message passing kernel (MPK).
  Communication Daemons are examples of these types of processes. *Query Drivers*, as defined below, can be also implemented with these kinds of processes.
- IPC_in / network_out processes (IINO). Those processes (bottom side of figure 4.2) receive messages by IPC mechanisms through the message passing kernel (MPK), and send messages to the network. As in the above case, the

arrows in the figure show only the main characteristic, which is the issue of requests into the network, but they can also receive answers, *e.g.*, acknowledgments.
An example of this type of process are drivers for managing external services, such as a *database service* or a *grid service*, which are in the network side, and which usually are servers waiting for messages.

- IPC_in processes only (IIO). These are processes whose principal characteristic is receiving requests -by IPC mechanisms- from local processes. These are useful for implementing tools responding to the request/response paradigm. See the left side of the figure 4.2.
  A *console* tool or a *monitoring* tool, are examples of implementation of these kinds of processes.
- IPC_in / IPC_out processes (IIIO). These are processes in the core of the system, which only know about local processes. They receive requests from, and issue also requests to other processes using IPC mechanisms (through the MPK kernel). See the right side of the figure 4.2.
  An example of this type of process are applications having local communication with the engine, as for example a *Web Portal*.

Message flow.

The flow of the messages (figure 4.3) goes from the top side through the bottom side of the figure, but, in between, messages can be pre-processed by processes in both the left and right side of the same figure. This means that a message gets into the system using an entity multiprocess of the type *NIIO*, and its process can deliver additional messages to other processes *IIIO, IIO*. These latter answer directly to the requesting process or may send a request into the network (through a IINO process) to another remote process (other engine or an external service).

Machines.

The *machines* are external components to an engine, which can be running in the same computer or in other one accessible through the network. In other words, a machine represents a service, a tool or an application, which is accessed by the local engine. A machine can be also a client issuing requests to an engine.

In general, we define client machines and server machines, depending on their functionality :

- Client machine. Any machine on the network, able to get in touch with a network_in / IPC_out (NIIO) process. This means that an external application (in the network side) issues requests to a distributed systems engine (DSE). An example of such a machine could be another engine, or a grid component which is looking for access to a medical image.
- Server machine. Any machine on the network, which could be reached by an IPC_in / network_out (IINO) process. So, a distributed systems engine (DSE) could reach external applications (machines) in the network side. An example of such a machine could be a PACS, a DICOM image server.

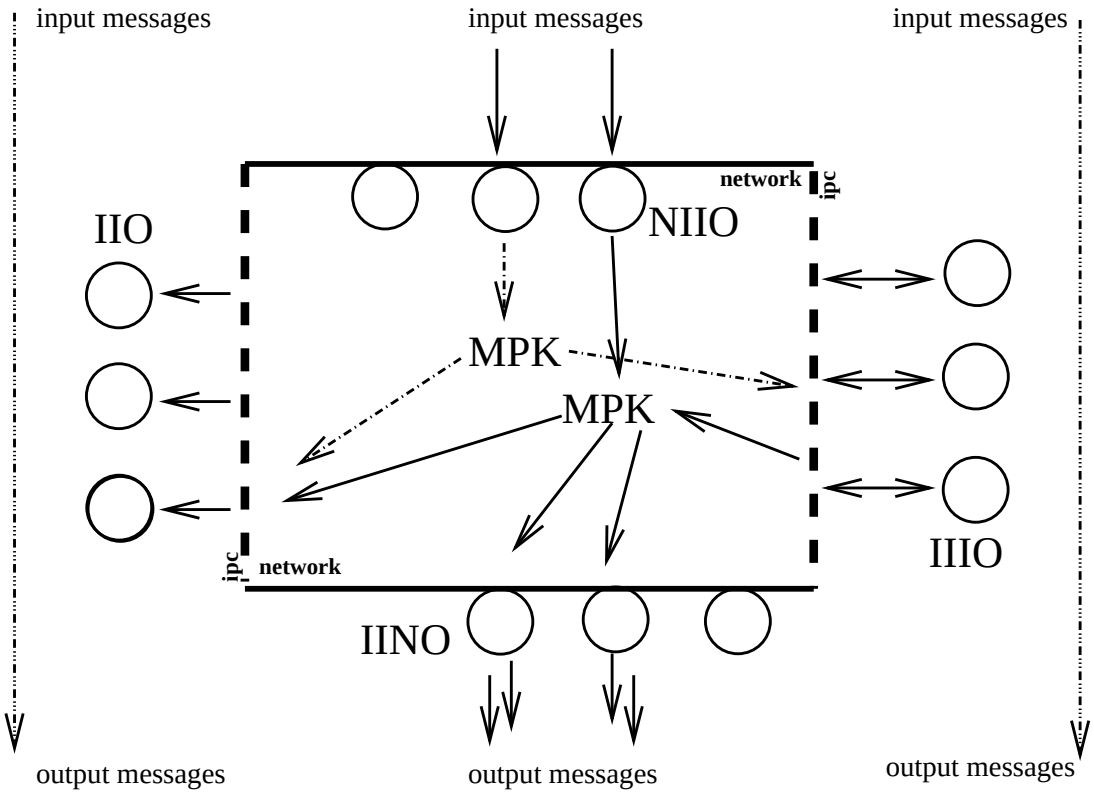


FIG. 4.3 – Message Flow.
The arrows represent the direction of the messages. In this figure there are two instances of the MPK.

An engine must manage different types of machines and different instances of each one; *e.g*, a machine type can be a DICOM server, and an engine must be able to have connection with different servers of this type, which means that the engine can manage a different DICOM server at the same time. In the same way, another engine is seen as an external machine.

Special processes.

Special processes are another type of external component to the engine, but running in the same computer and also having an IPC communication with the engine.

They are also client and server processes as follows :

- Server processes. A process of type IPC_in / IPC_out (IIIO) is able to receive an incoming messages from a distributed system engine, and produce a response.
- Client processes. A process of type IPC_in / IPC_out (IIIO) is able to issue a message into a distributed system engine, and it then waits for a response. As an example, a *Web Portal* could be an external application, running in the same host of the engine, and having local IPC communication with the engine. This *Web Portal* can be a multi-process application, which has *Client*

Processes and *Server Processes* to the engine. The former ones make requests to the engine, and the second ones answer to requests from the engine (*e.g.,* status processing).

What makes the difference between machines and special processes is the kind of communication between them and the engine ; while a machine always has a network connection with the engine (*e.g*, sockets), the special processes use IPC communication. They are two different kinds of components which can interact with an engine, whatever their functionality. For example, an engine can interact with a local cache service by using IPC mechanisms, but it can also have a remote communication with a data grid service for storing sets of images, which is also a kind of cache for the engine. In the first case, the cache service is a set of special processes, in the second, it is a machine. This versatility is mandatory when dealing with external components. Remember we do not have control on the development of them : they are as they are.

Transparent machine and processes connections.

The architecture enables the connection of machines (see figure 4.4) as follows :

- Client machines and server machines.
  An external machine gets in touch with a NIIO process in order to request a message processing which, transparently, is delivered to a server machine for processing. This may imply to make a pre-processing of the messages before accessing that server machine. See figure 4.4-i ; *e.g*, a remote engine (client machine) queries another engine for getting access to an image in one hospital (server DICOM, or server machine)
- Client machines and server processes.
  An external machine sends a message to a NIIO process. This message is transparently issued to a server process for processing. See figure 4.4-ii ; *e.g*, same as the example above, but the engine looks for the image in the cache service instead of the remote DICOM server. We assume that our cache tool uses only IPC communication, so it is a special process.
- Client processes and server processes.
  An IIIO process sends a message to another IIIO process in order to collaborate. See figure 4.4-iii ; *e.g* ; a security tool (client process) asks for a cache service (server process). We assume that both have IPC communication.
- Client processes and server machines.
  An IIIO process sends a message to an IINO process in order to access a server machine. See figure 4.4-iv ; *e.g*, a security tool asks for access to a database service, which is a remote server machine (*e.g.,* MYSQL daemon).

4.4 DSE[1] : Transaction Layer

At this layer a *Distributed System Engine (DSE)* is defined as a set of *Drivers* (defined above) :

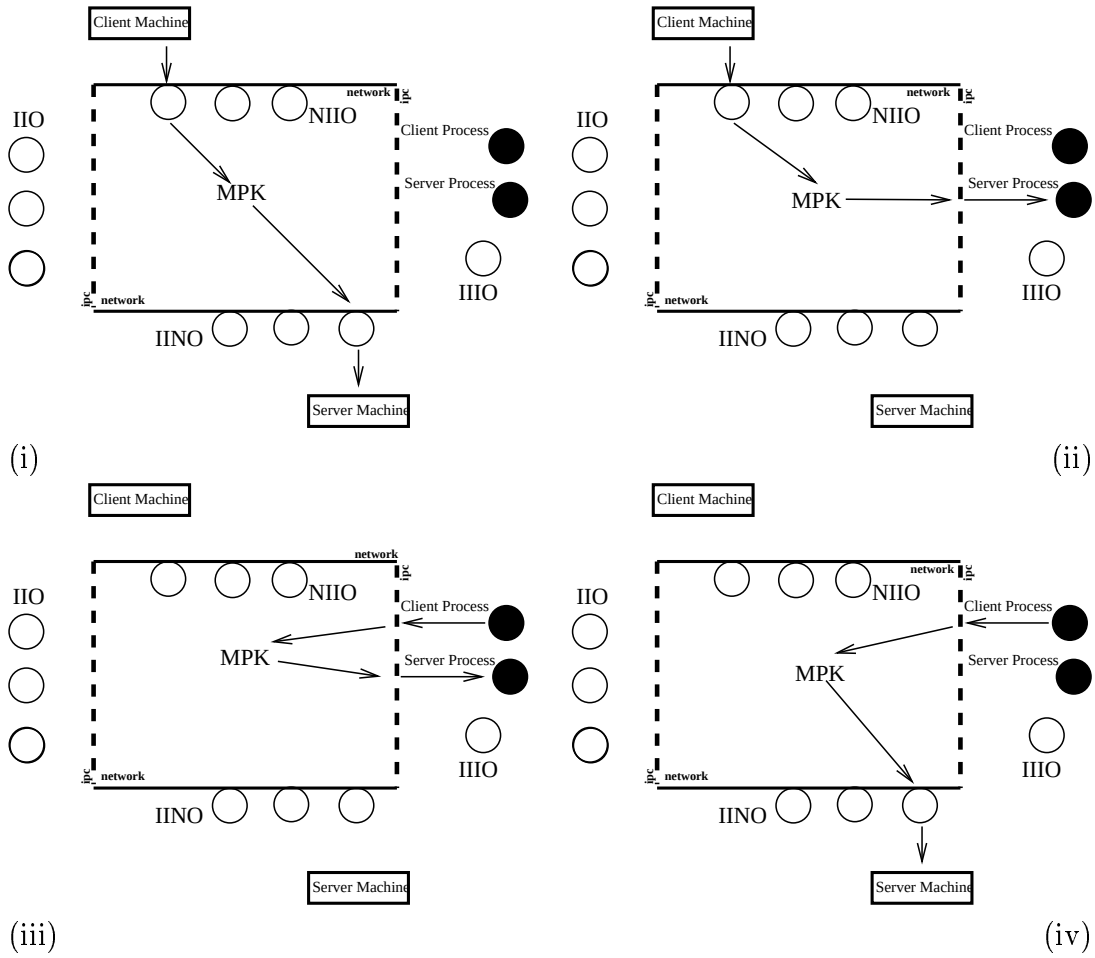


FIG. 4.4 – Machines' connection

(i) Client Machine with Server Machine , (ii) Client Machine with Server Process, (iii) Client Process with Server Process, (iv) Client Process with Server Machine. The black circles represent special types of processes : server and client.

DSE[1] ↔ {Drivers}

On top of the simple message passing layer, DSE[1] defines atomic operations (transactions) made of multiple sub-operations. A transaction succeedes if, and only if, all sub-operations succeed. If a failure occurs, the system must be left in a coherent state. It must offer the ACID properties : Atomicity, Consistency, Isolation and Durability [1].

We deal with three types of transactions (see figure 4.5) which we call *Queries*, *Tasks* and *Requests*. This allows us to deal efficiently with the complexity of transactions which involve multiple calls to external services and engines. *Queries* are a set of *Tasks and Requests* which can be executed in sequence or in parallel. Similarly, *Tasks* are a set of Requests executing concurrently to shorten the processing time. *Requests* are a set of sequential messages to a service in the network side. *Queries, tasks, and requests* are managed by a special type of processes that we call *DSE Drivers*, but for simplicity in this document we will refer to them by using just the

term *Drivers*. A *Driver* is a process (or a set of processes) handling different kinds of transactions instead of single messages.

The processes which were described in layer 0 (DSE^0) are the base for layer 1. Drivers are multi-process communicating entities, so a set of processes of layer 1 have the highest semantic significance in this layer (DSE^1) and become *drivers*. Figures 4.6 also represent the local engine, but are composed as a set of interacting drivers, depending on their type as is described below. Additionally, the figure 4.8 shows the differences between the different layers.

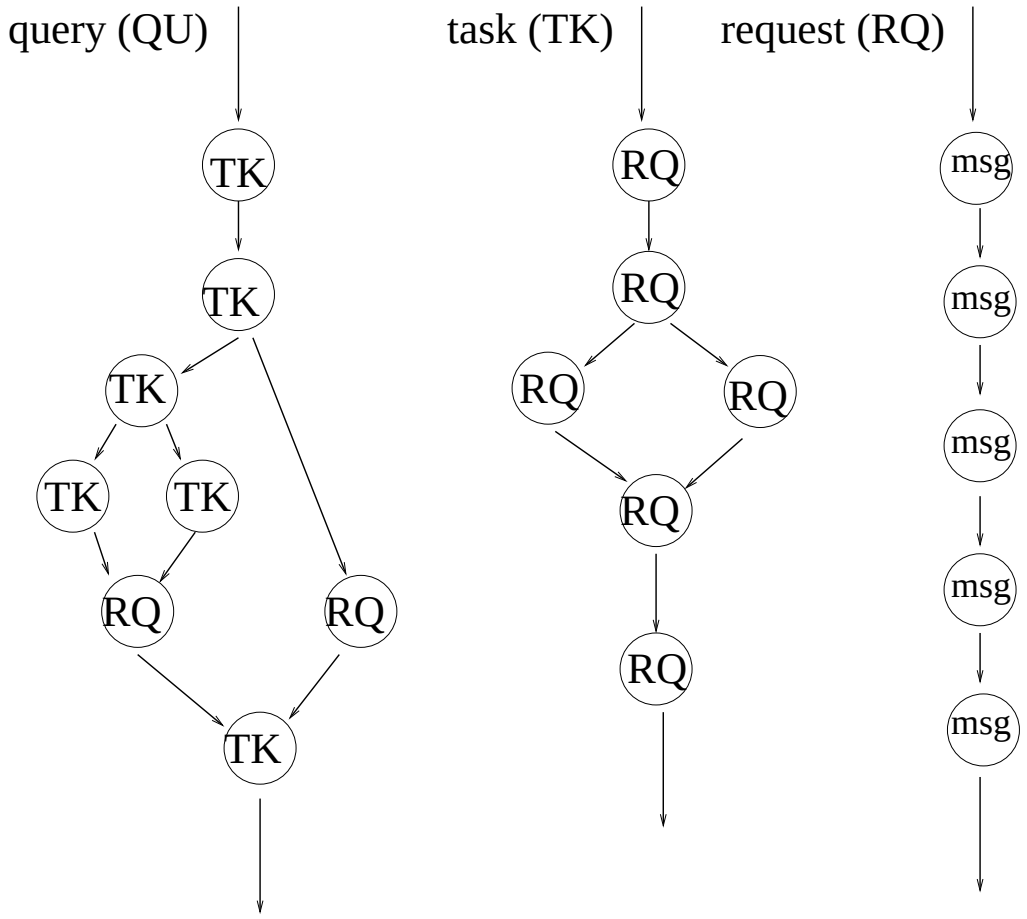


FIG. 4.5 – Transactions' types

(i) Query : a set of tasks and requests, (ii) Task : a set of requests, (iii) Request : a set of messages to the network. The circles represent what compose each type of transaction, and the arrows represent the order of execution. Thus, splitted arrows represent execution in parallel, and raw of arrows represent sequential execution.

4.4.1 Driver types

– QUeries Drivers (QUD) are processes managing a whole transaction (query) made up of a set of *Tasks and Requests*. A *query* could imply concurrent or sequential access to different external services, or just accessing local tools. The tasks which compose the query, are solved by a *task driver* (described below),

and the requests by a *request driver* (also described below). This means that for solving a query, a QUD must access (by delivering messages) other drivers in the same engine, even if the solution of part of the query complies one to access an external service in the network side.
In this way, an access to a TKD or RQD depends on the nature of the query, and is not mandatory. A query could be implemented as accessing a TKD but also having direct access to a RQD. This means that a *query* could be implemented without *tasks*, but only with requests. Meanwhile, the QUD can access the available tool drivers in order to execute local functions.
For example, a query of a 3D image needs to localize, transport and assemble the image files into an unique file (an image can be represented as a set of DICOM files or slices). So, we can define a task for getting the image (localization and transport), and another for assembling the image. This first task is transparent and solved by a TKD. The second, can be solved by using a local tool.

- TasKs Drivers (TKD) are processes in charge of a specialized part of a query (or task). This could imply one must get a parallel access to an external service through a set of request drivers. TKD offers parallelism, distribution and transparency as a service which could be used by a QUD ; *e.g*, a QUD requests a file to a TKD, but it does not check the localization of the file : the TKD can get it from the cache, from a local hospital, or from a remote hospital (accessing another engine) in a transparent way to the QUD. In order to find the file, the *TKDs* uses the *RQDs* available in the engine.
- ReQuest Drivers (RQD) are in charge of accessing remote components such as other engines and external servers. They solve low level issues such as connection management. These drivers transmit messages and receive responses that they edit before sending them to the calling processes [5].
  For example, a RQD can have access to a database service (Spitfire [138], MYSQL), so it manages the connection, the session opening, and uses the paradigm answer/response in order to solve the transaction. In these cases, the request and its response are single messages sent each way, so the response to the TKD or QUD is basically the same response message. In contrast, in the case of a RQD which deals with a DICOM server, a request is composed of a set of messages, as is described in the DICOM protocol [49] ; so the driver must solve the whole request before answering to the calling driver.
- TOol Drivers (TOD) are processes performing internal operations that are independently implemented for reasons of performance and modularity. Examples of such processes are the caching of requests, files and results, logging, security checking, image processing and manipulation of console operations. Tool drivers can be accessed from QUD, TKD and RQD.

These *drivers* use the *processes* of the lower layer and coordinate them in order to develop higher level functionalities and semantic significance.

[5] In order to improve concurrency and to allow an easy modular software development, there is a 1 to N relationship between the processes of QUDs and TKDs, and similarly, a 1 to M relationship between the processes of TKDs and RQDs. This means that a query can be split into N tasks, and that a task can access in parallel an external service through a multiprocess RQD

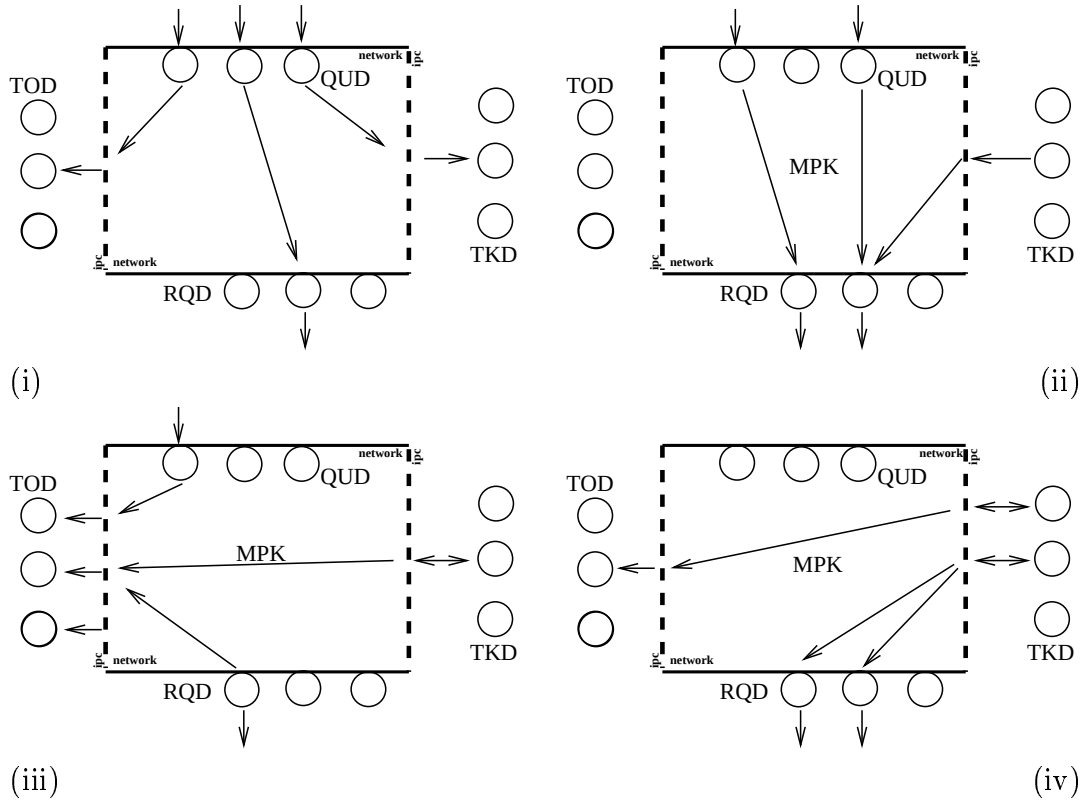


FIG. 4.6 – Drivers
(i) Query Driver (QUD), (ii) Request Driver (RQD), (iii) Tool Driver (TOD), (iv) Task Driver (TKD)

An application developed as an engine works as follows : (i) A message is received by a query driver and a query is initiated, (ii) The query starts different concurrent tasks, using independent processes (TKD), (iii) Each task access the requests drivers (RQD) so that it can reach the external services, (iv) The request drivers (RQD) open connections and send messages to the external services, (v) Each driver uses the tools it needs (TOD).

Revisiting the examples described above, we can summarize that a client machine sends a query to the engine (*e.g*, get a 3D DICOM image), which is received by a QUD. The query driver gets the slices (which compose the image) through a TKD, and once it retrieves them, it uses a TOD for assembling the slices into a single image. The TKD uses different RQD for solving the task : (i) accesses a database service through a MYSQL RQD in order to get the localization of files, and them uses the DICOM RQD in order to get the files (slices) from a DICOM service. An application of this type will be analyzed in detail in the next chapter.

4.4.2 External Applications.

External Applications are applications which run on the same host as the engine and which have IPC communication facilities with the engine instead of network communication. They are composed of *special processes* as were defined in section

4.3.2 ; therefore, *external applications* are the next level of semantic significance for the *Special Processes* from the layer 0.

These applications are also independent of the engine, which means, we do not have control over them, the only thing we can do is to interface them, and the interface they offer is IPC communication. They are also divided into server and clients.

- Server applications.
  External processes to the DSE but located on the same host, which are able to receive incoming messages from a distributed system engine, and produce a response. A server application is based on IPC_in / IPC_out processes which could start interaction with other processes into a DSE.
- Client applications.
  External processes to the DSE but located on the same host, which are able to issue a message to a distributed system engine, and wait for a response.

In section 4.3.2 we have proposed a cache as an example of *special processes*. What really happens is that the engine sees an *external application*, the cache, which is accessed by IPC mechanisms, and which is composed of *special processes*. In our example, this cache is a multi-process entity.

4.5 DSE^2 : Distribution Layer

At this layer a *Distributed System Engine (DSE)* is defined as the *union* of different sets of *Internal Tools*, *Services Drivers* and *Services Daemons* :

DSE^2 $\leftrightarrow$ {Tools} U {Service Drivers} U {Service Daemons}

DSE^2 brings distributed facilities on top of the two lowest levels. This layer adds semantic functionality to the transaction layer (DSE^1) in order to convert classical atomic transactions into distributed transactions, which also have the ACID properties, but additionally are spread over a distributed system.

The main objective of this layer is to offer collaboration between different *DSE* in order to build a Distributed System with characteristics as :

- Concurrency : It increases the efficiency of a *DS*. True concurrency comes from the separate activities of users, the independence of resources, and the separate location of server processes in the system.
- Transparency : It conceals the users and the applications programs from the separation of components.
- Location Independence : Users can retrieve and update data independently from their storage site.
- Distributed Query Processing : Users can query information residing on another node. The query is executed at the node where the data is located.
- Distributed Transaction Management : A transaction can update, insert or delete data from multiple sites. A transaction in this layer has properties of distribution while in DSE^1 it is local.

– No reliance on a central site : All sites are treated as equals. Each site owns its data.
– Local Autonomy : Data are owned and managed locally. Local operations remain purely local. One site in the distributed system does not depend on another site to run successfully.

The distribution layer is in charge of localizing data and services (sometimes resources), transmitting requests to proper hosts, collaborating between DSE engines, etc.

However, a DSE aims at taking advantage of external resources such as those provided by a Grid (specially storing and computing). In this way, a Grid is a source of resources for a *DS* composed of DSEs. A Grid aggregates resources, across multiple domains and offers resources sharing to its users.

We talk about Grid resources in the sense that they are huge, vast, and expensive. Usually, a *DS* has that kind of resources but only for limited or local use ; thus, having access to the Grid resources becomes usefull.

A Grid can become, then, a partner to our *DS*, in the sense that it provides high computing and huge storage capacity (if required). A DSE must provide an interface to the Grid in order to have access to those resources, and such an interface can be developed as a *Driver Service - SDR* (described below).

4.5.1 Components

This layer defines new components *SDA, SDR and Tools* (described below) by using the ones (from the layer 1) which deal with network communications *(Query Drivers, (QUD) Requests Drivers (RQD))*, and the Tool Drivers (TOD). We will refer later in this document to *internal services*, when talking about the SDA (Service DAemons) and SDR (Service DRivers). The figure 4.8 shows the differences between the different layers.

Service DAemons (SDA)

A DSE offers services to *Client Machines* on the network side, and these services are accessed by issuing *Queries* to the *Service Daemons (SDA)*. A Service Daemon is composed of 1 or many Query Drivers (QUD) of layer 1 (see figure 4.7), and deals with all the problems related to accessing the service : (i)Communication transport protocols as HTTP, XML and others (ii) Security, (iii) Access to the service, (iv) Execution of Queries.

The *Service Daemon* can be a Grid Service, a Web Service, or simply a Daemon which offers a special service to a set of users ; *e.g*, a SDA can offer the service of hybrid queries, but it can be composed of different RQD in order to manage different phases or possibilities in the solution of such a query. For example, each one of three RQD can accept queries for three different protocols of messages (proprietary, SOAP, OGSA), even if the query is the same and the solution will be the execution of the same transaction. This enables the engine to solve the query by accesing three different external services, each one having different interfaces. The fact that one has three RQD does not mean that there are three services : there is only one service implemented with three RQD.

Service DRivers (SDR)

A SDR is a group of RQD in charge of solving different possibilities of a request, by accessing *external machines*. A DSE in its internals could need to access external services for getting data or for using resources (store and computing), but the diversity of services can imply the use of different RQD and SDR.

For example, an application can manage different image formats such as DICOM3, NEMA [6], GIF, INR, or even a RAW format. So, it can be defined an SDR for offering an *internal service* (to the engine) of *image retrieval*, which can be composed of two request drivers, the former for managing the protocol DICOM3, and the second for FTP [7]. This Service Driver goes into the network in order to contact the appropiate external service (DICOM, FTP).

In our example, although there exist two RQD, they enable only different ways of solving the same request : to *retrieve an image*. This is different than the possibility of having different instances of the same RQD (layer 1), which represent connection with different server machines (layer 0). This means that each one of several RQD can have a different instance dedicated to the connection with each one of the DICOM servers (hospitals). However, there is only one SDR. Another example of SDR can be the *database service* ; it can be composed of different request drivers depending of the database engine, *e.g.*, one RQD for Spitfire [138] [8], and another one for MYSQL ; or even, Grid SDR, having a RQD for DataGrid/EGEE, and another for MicroGrid [9]. Similarly, different services of a Grid, such as Storage Element (SE), Replica Catalog, etc, can be implemented as different RQD of the same Grid SDR.

It must not be confused with the task drivers (TKD) of layer 1. A TKD can access different SDR in order to solve its task. For example, when solving a task of *getting an image*, there are at least two requests to solve : *localizing the image, and retrieving the image*. Both requests are solved by a different SDR, the first one uses the *Database SDR*, and the second one the *Image Retrieval SDR*.

Tools

A DSE must be able to perform the internal operations of the low level for *checking security, cache, and localization of data and servers*. For reasons of performance and modularity these must be implemented as local independent services which we call *Tools*.

Actually, these functionalities involve different layer components :

– File caching : Low level cache functions in order to improve the latency of accesses. We see this tool as something which can be implemented by using processes (IIO) of the lower layer (0), which can be accessed and requested by different processes in the engine, even if there is no transaction functionality implemented.

[6] An old format, before becoming DICOM

[7] We suppose that we can transfer the other images by FTP.

[8] Spitfire is the Database Service offered by the DataGrid project.

[9] Microgrid is a lighweight grid, developed for the Medigrid Project [131], which we have used for testing.

- Request caching : The idea is to have a cache of the answers for previous requests, instead of only having files and images. We consider this as a higher level problem, so transaction properties are needed. Collaborative cache mechanisms [46] [122] require to get provided with distribution facilities, so a such kind of cache must be designed and implemented at the *DSE*^{2} *level*.
- Security : Access control, authorization granting, denial of service, secure transfer, integrity checking, authentication, etc. Most of these functions can be implemented as basic low level services (layer 0), but others (authentication) can require one to access a server with certificates, so a TOD (layer 1) could also be useful for managing transactions, while a third layer tool could provide distribution.
- Data Localization : Functions for distributed data localization, in order to get interactions between different DSEs. Engines can take advantage of completely decentralized Peer-to-Peer (P2P) techniques [2] [140] or of semi-hierarchical tree structures such as LDAP. It depends on the implementation, but a tool for localizing files in other engines is only needed when there is interaction between engines of the same type, *e.g*, two DM^2 engines, as is shown in the next chapter. This type of localization tool is in the third level, while others, as the one described above (a TOD, section 4.4.1) aim at localizing only files in the internals of the engine. For example, an engine can have registered (metadata) images from many local hospitals (in a geographical region), so it is able to localize those files by using its internal database service. To localize the same file from another engine, is quite different. A tool of the distribution layer will localize the engine which knows where the file is stored, instead of the file localization itself. In other words, each engine is responsible for localizing the files in its local space and to offer that service to the other engines.

In summary, a *Tool* may be composed of tool components of the different layers. For simplicity, we will refer to a *Tool* as a layer 2 component having components of lower layers. Remark : tools as cache and security are being developed by other groups of our researching team [187] [46] [47] [48] [90] [123] [122] and will not be studied in this document.

Figures 4.7 and 4.8 show the differences through the 3 middleware layers. The vertical progress shows aggregation of entities in order to add semantic significance, for example many Processes are grouped in Drivers, and many Drivers are grouped in Service Daemons and Service Drivers. The horizontal view in figure 4.8 is a representation of the interaction of elements in each layer.

4.5.2 Schemas

A set of *DSE* can be organized in hierarchical or decentralized schemas. The first approach allows one to define fixed connections between DSEs, with an implicit relationship of dependency. The second, brings independence from a central server : it refers to P2P techniques. However, a mixed schema is more realistic.

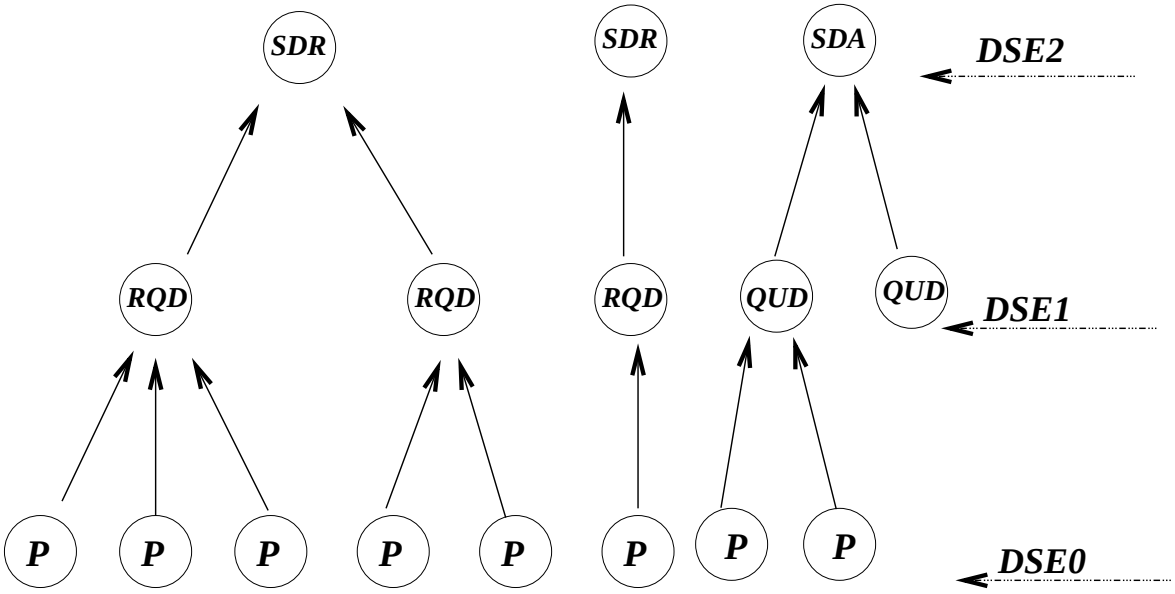


FIG. 4.7 – Verticals' view comparative of layers 0, 1 and 2
Aggregation of Processes and Aggregation of Drivers. Multiple Processes are grouped in Drivers, multiple Drivers are grouped in Service Daemons and Service Drivers.

Figure 4.9 shows examples of these two architectural frameworks, and considers the permanent availability of computing and storing services in a GRID, which are interfaced with the engines.

4.6 DSE^3 : Application Layer

At this layer, a *Distributed System Engine (DSE)* is defined as a set of *Services* :

DSE^3 $\leftrightarrow$ {Services}

A *Service* is an independent application, with a known interface, and which performs a function for the users; thus, a *DM^2 Service*, is one of the services offered by a DM^2 Application. We refer to this concept meaning functions of low (data access) and high level (queries by content, hybrid queries) offered by the DM^2 Application to their users.

This layer must offer programming interfaces (API) so that end-user applications can be built on top of the underlying distributed system.

This layer becomes the core of a specific application, built as a Distributed System and which has vast requirements of processing and data storing.

The design of the application is based in the concepts of the previous layers, so :
– It is a DS built as a set of interacting *DSEs (engines)*.

- Each Engine is composed of multiple processes which represent *Drivers (QUD, TKD, RQD).*
- The application develops *Queries, Tasks and Requests* (three types of transactions), instead of *Drivers*. The *Drivers* are used to execute the transactions that use lower level API. Those *Drivers* and APIs are offered by the middleware layers.
- Each *engine* is named with the application name, *e.g.*, the application DM^2, which we will describe in the next chapter, is composed of *DM^2 engines.*
- Each *application engine* looks for tools (*TOD*) internally, for services (*SDR*) in the network and for resources in a partner Grid (also a *SDR*).
- The application also offers services, which in its internals are processed as *queries transactions*, and executed into multiple *RQD (SDA)*

The applications are developed over the functionality, *tools and drivers* offered by lower layers (0, 1, 2). However, the developed application must be designed using the proposed architectural framework (as described above), and being separated in server and client components. For example, an application as DM^2 (described in the next chapter) has a lot of tools and internal services (SDA, SDR), which are developed by implementing different kinds of transactions (queries, tasks, and requests), and which run into different types of Drivers (QUD, TKD, RQD, TOD) ; but, all this, corresponds to a server implementation, called a *DM^2 engine* : this is the third layer. The next layer, the forth, is the client side of the application, the one which deals with the user issues.

In other words, a *driver* manages a special kind of *transaction*, but the *driver* and the *transaction* are different things ; for example, the *QUery Drivers (QUD)* manage *queries*, the *TasK Drivers (TKD)* manage *tasks*, and the *ReQuest Drivers (RQD)* manage *requests*. The middleware layers (*e.g., DSE^0, DSE^1 and DSE^2*) implement those different kinds of drivers (along with the *service and daemons drivers*), while the application layer (*e.g., DSE^3*) implements the transactions : *queries, tasks and requests.*

4.7 DSE^4 : User Layer

The forth layer is defined as a set of *interfaces* to an engine.

$DSE^4 \leftrightarrow$ {Interfaces}

The *User Layer* is in charge of offering high level access to the services implemented by the application engine. Thus, for our medical application, this layer gives access to the services offered by the *DM^2 engine.*

At this level, transactions become to its highest semantic level (*e.g*, Hybrid Queries). Users only know about *Queries*, they do not know details of execution of these queries, such as distribution, complexity, and magnitude of the resources involved (computing, storage, network).

This level can be implemented as graphical interfaces (GUI) or as client applications which use the APIs offered for the precedent layers. In this way, *Users* can

have access to an engine service.

This layer is the client side implementation of an application engine ; for example, the *DM*^{2} *application* offers the API0 and the API3, which are useful for developing the client side applications for the users.

4.8 Discussion

4.8.1 Extensibility and Scalability

The proposed architecture (DSE) allows one to build distributed systems (DS) which are highly extensible and scalable, because of its ability to add tools and services by using its structure of *drivers* and internal services.

The structure of *Drivers* allows a DSE to be open and easily extensible **(openness)**. By defining new *Query Drivers (QUD)*, an *Engine* can offer more complete services (*e.g.*, different access protocols), by adding new *Service Daemons*, the Engine can offer more different services to its users (*e.g.*, data access, metadata storing, image query by-content). The definition of new *Request Drivers (RQD)* also enables the engine to have access to external services by managing different protocols ; the addition of new *Service Drivers (SDR)* gives the Engine access to multiple external services (Grid, DICOM, database engines). Similarly, the engine enlarges its internal autonomy by having access to more and different *Internal Tools*, which are empowered by adding *Tool Drivers (TOD)*.

The division of transactions into different types (queries, tasks, requests) eases the decomposition and distribution of complex requests (as an hybrid query) ; especially, the concept of *task* and the drivers (TKD) to deal with them, which enables the development of the distribution layer, by easing transparency and independence.

The structure of multiple processes enables parallelism and concurrency in order to improve performance. Performance can be also improved by having multiple instances of Drivers.

Resources can be added by adding more concurrent Service Drivers (SDR) that get access to shared Grid resources. A DSE aims at taking advantage of vast external resources as those which are offered by a data or computing Grid. Adding more resources, the system should be able to handle a higher quantity of requests (queries). Additionally, by dispatching more concurrent instances of the drivers, the engine is enabled to deal with a larger number of concurrent transactions **(scalability)**.

Software interfaces (APIs) must be specified, documented and available to developers. By using those APIs, an external system can access the services offered by the engine [10].

The middleware layers can be used to build different applications. As an illustration, figure 4.10 shows two different applications, implemented as two different types of engines : DM^2 and Cache.

[10] Two prototype APIs, for layers 0 and 2, have been developed and are presented in chapter 4

4.8.2 The DSE architecture VS our Medical Image Manage Problem

Before beginning this research, we considered the problem of systems for Medical Image Processing containing vast archives of raw images (as was presented in chapter 2), and we also considered its main constraints. Many of these constraints have a direct relationship with the classical theory of distributed systems, such as openness, scalability, security, extensibility, concurrency, and location independence. Others are closest to ongoing research in grid computing (data-intensive computing and storing, resource sharing). In addition, other constraints are connected to medical applications (confidentiality of medical data, queries by content, structured hybrid queries, restriction of the replication of data (see chapter 2) and the diversity of image processing algorithms to apply to the medical images.

Because of the complexity of the problem, we decided to define an architecture that proposes to build a system from its conceptual definition. Rather than starting with the development of a system, we proposed a concept of DS and then we tested it with a prototype (presented in the next chapter).

The architecture (*Distributed Systems Engines (DSE)*) addresses our problems of structure, extensibility and distribution, as has been shown in the previous section, but it also addresses our medical concerns :

- Tool Drivers offer the possibility to implement functionalities of security for medical applications. Thus this concept has allowed us to develop in parallel, an additional prototype system for security with medical images. Another research group of our team [187] works on this topic and its product will be integrated as a *Tool*. So developed applications can open and close their doors when they want. These applications can also access external services and resources, while offering their services to a restricted group of users.
- Medical images are composed of sets of files rather than single files, *e.g.,* a cardiac 3D MRI image may be composed of 300 files (and more). The Drivers structure allowed us to increase the performance while processing these images, by implementing transfers and processing them in parallel. Performance was also improved by basic Tools (layer 2) such as the Cache Tool (another research group of our team [187] works on improving cache techniques).
- Large storing capacity will be provided by Data and Computing Grids. Our architecture considers a Grid as a natural partner, and the structure of Service Drivers (SDR) and Service Daemons (SDA) allows interfacing with Grids (*e.g.,* to store anonymous images) in a transparent way.
- Queries by content over medical images archives are CPU intensive, so the computing resource becomes critical. The solution, again, is the access to Computing Grids through Service Drivers. Task Drivers will help in providing transparency and parallelism in the access to the computing resource.
- This kind of system requires managing metadata about the images. Query Drivers allowed us to define and access multiple databases.
- The databases of medical images will be used for medical diagnoses and research. For that, the system must be open to process images with different algorithms, which are registered in the system. Services Daemons (and APIs)

are useful to deal with registrations of algorithms and acceptation of hybrid queries, while Task Drivers, Tool Drivers, Request Drivers and Service Drivers are useful to deal with the processing of those queries (*e.g.*, a hybrid query over many images).
- Tool Drivers are useful to implement *Tools* for dealing with basic image processing functions, such as conversion of formats, extraction of characteristics, etc. Such a *Tool* can be accessed from different processes as a local service.
- While designed with a idea of medical large scale application, the DSE framework is generic and can addresses most types of applications.

–

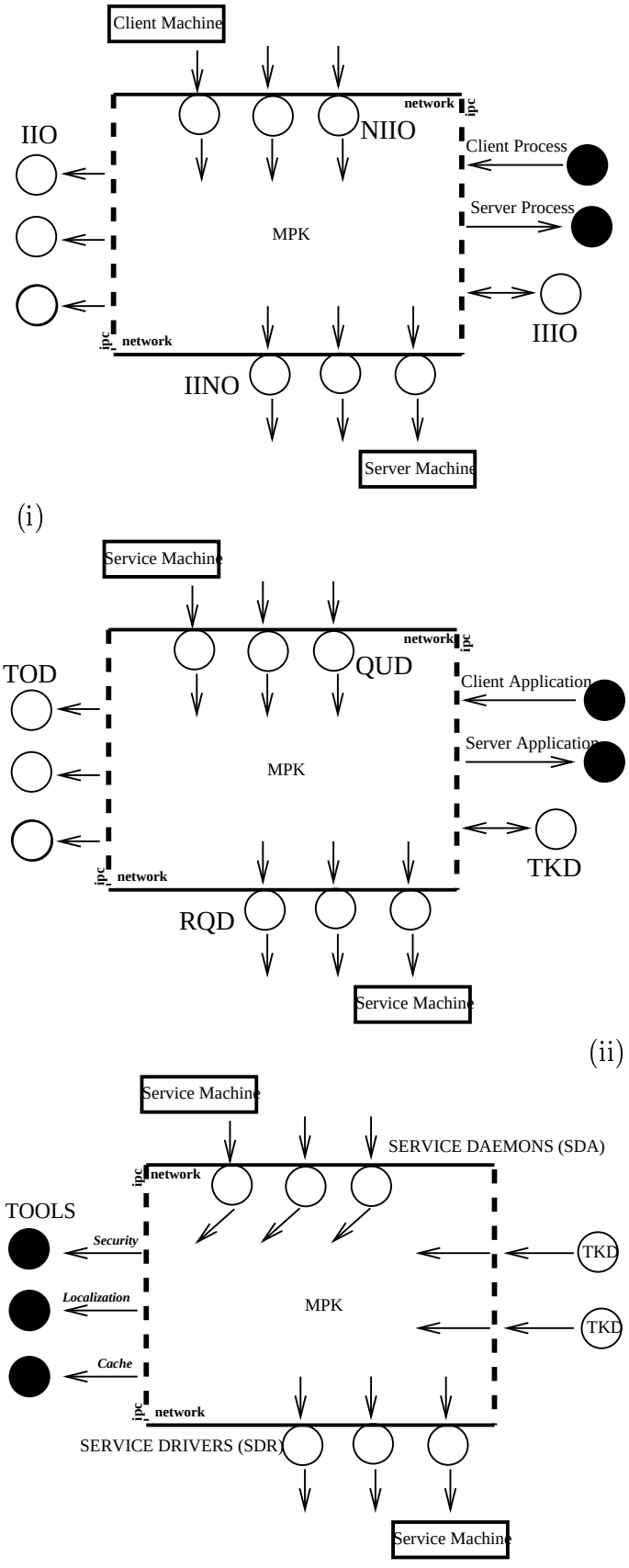


FIG. 4.8 – Horizontals' view comparative of layers 0, 1 and 2
(i and ii) Processes at layer 0 become Drivers at layer 1. Special Processes become Applications. Machines become Service Machines. (ii and iii) QUDs become Daemons, and RQDs become Service Drivers

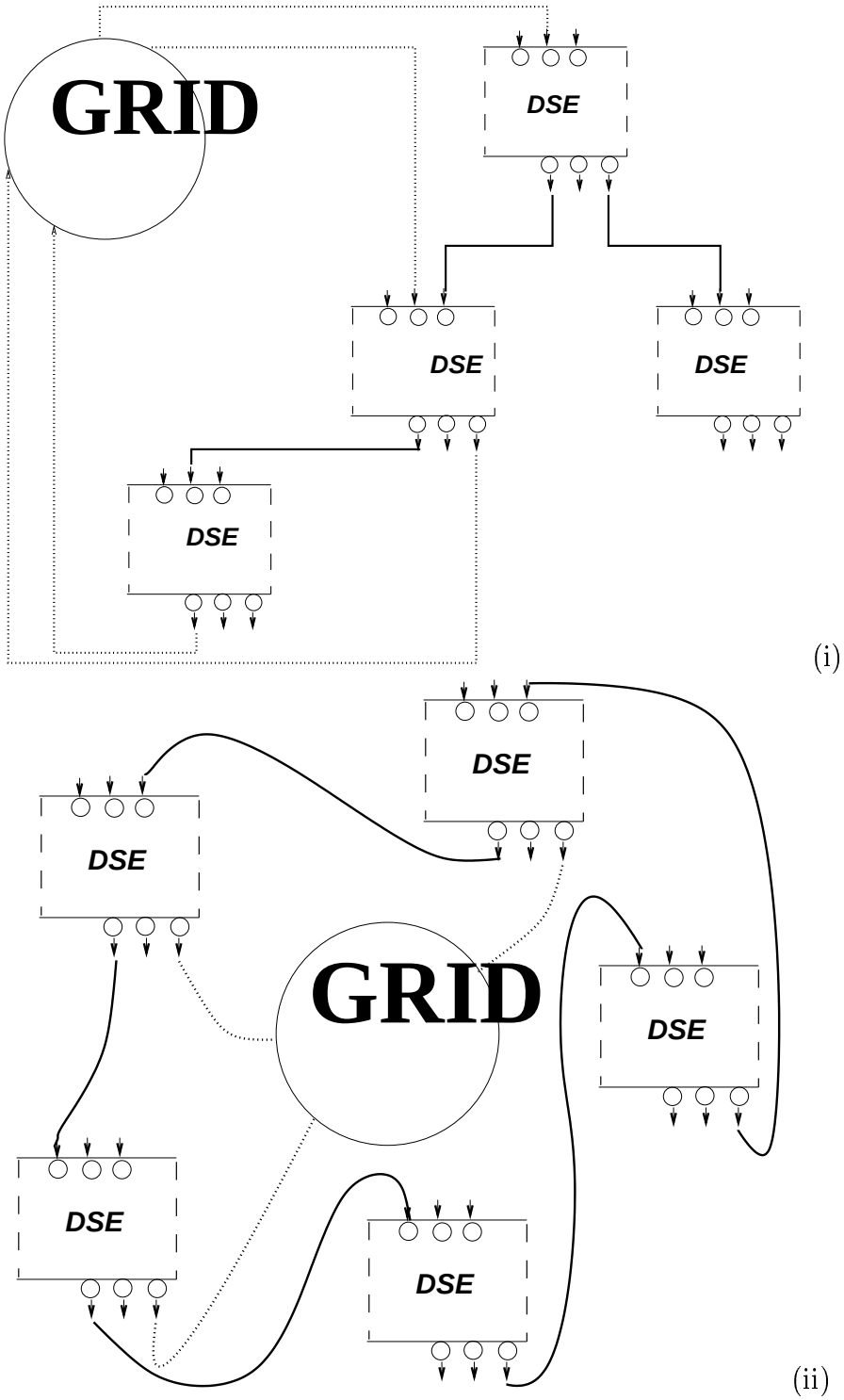


FIG. 4.9 – Distributed System

A DS composed of a set DSEs and Grid services. (i) hierarchical, (ii) peer to peer, (ring) topology. The Grid participates as a resource provider, and is not, in any case, a router of transactions. Hierarchical and P2P relationships are represented with continuous lines, while connections to the Grid are dotted lines.

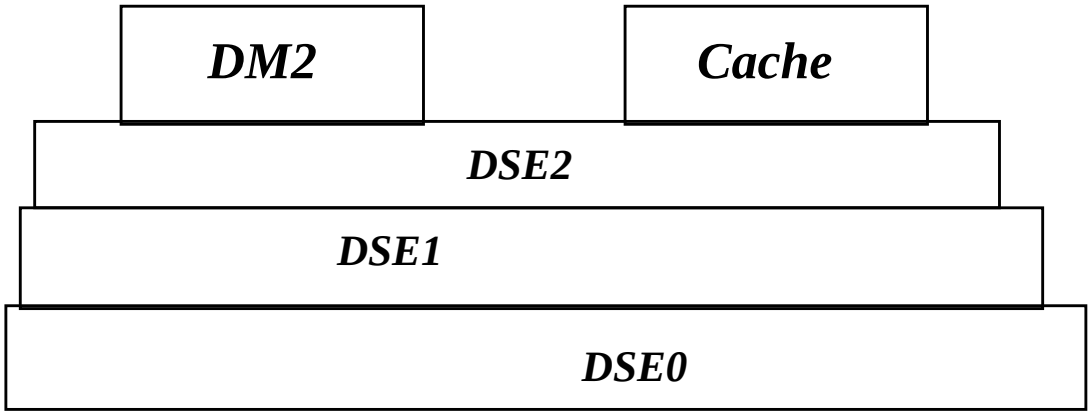


FIG. 4.10 – Illustration of two applications using the architecture
(i) DM² Engine. Levels 3th and 4th are the DM² application. (ii) Cache Engine. Levels 3th and 4th are a cache application.

Chapitre 5

Implementation

*"Data resources are grid-enabled by deploying low-level middleware systems on them", **Rajkumar Buyya**, University of Melbourne, Australia*

–

Summary 5

*We present a prototype system which is an implementation of the multi-layered architecture (**Distributed Systems Engines - DSE**). With it we have developed a medical application for managing n-dimensional medical images querying and retrieval, and also highly semantic structured queries (content based and hybrid queries) over them.*

*First, we address the middleware layers, and then the application layers of the DSE architecture. Our middleware prototype implementation is called **Distributed Systems Engines Manager - DSEM** or **engine**, and the application prototype is called the **Distributed Medical Data Manager (DM^2)**. The DSE/DSEM eases the development by enabling the access to large computing and storing resources in a grid.*

The engine (DSEM) is implemented mainly of an API for message passing (API0), a Message Passing Kernel (MPK), a dispatcher, and a tool for monitoring. It also has the ability of being connected with several databases at the same time.

*The **engine (DSEM)** offers internal services such as routing of messages, and the processing of transactions and files localization, whereas the **Distributed Medical Data Manager (DM^2)** provides services such as queries by content, hybrid queries, and querying and retrieval of sets of medical images.*

*DM^2 uses a grid as a computing resource. We have implemented DM^2 as a set of basic applications, called **Packages**. Each **Package** implements a service provided by the medical application.*

*The DM^2 is implemented as a set of **Packages** and the API3 which delivers remote queries to a DM^2 Engine. The **Packages** included with DM^2 are : (i) DM^2 core, (ii) DICOM, (iii) Relational Database, (iv) Grid, (v) Images tool, and (vi) Cache tool.*

–

5.1 Sketching Our System

In this chapter we present an implemented prototype of our system. The First section (5.2) addresses the middleware layers, whereas the second (5.3) studies the application layers.

First, we have implemented a middleware prototype, called **Distributed Systems Engines Manager (DSEM)**, which addresses the layers [1] 0 to 2 of the *DSE* architecture (chapter 4). The prototype offers internal services as routing of messages, management of transactions, file localization, and others capabilities which will be described below.

Next, in order to fulfill to the requirements described in chapter 2 and the medical usecases, we have developed a *medical application* on top of this middleware. This medical application is called the **Distributed Medical Data Manager (DM^2)** and addresses layers 3 and 4 of the architecture. It offers services over a large medical images data set, as queries by content, hybrid queries, querying and retrieval of sets of medical images, etc.

The DM^2 system is designed as a complex system involving multiple grid service interfaces and several interacting processes geographically distributed over a heterogeneous environment. It is an intermediary (proxy) between the grid and a set of trusted medical sites, which use the grid as a source of computing resources. To tackle the DM^2 complexity, we proposed the multi-layer architecture as outlined in chapter 4. We also analyze the interfaces between the DM^2 and underlying grid services.

Section 5.2 describes *DSEM*, and section 5.3 describes DM^2. In the next chapter we will analyze the *DSEM* performance, and present one application using DM^2.

5.2 The Distributed System Engine Manager (DSEM)

Our prototype is composed of *an Engine* and *an Application* using it. The *engine* is an implementation of the middleware layers of the *DSE* architecture, and is mainly composed of processes and drivers which interact at the lowest level by exchanging messages. An API for interfacing the engine is also provided. We describe below these components.

DSEM offers the following services :

- routing of messages between processes or entities of a higher level, such as drivers and internal services
- solution of transactions of three types : queries, tasks and requests
- basic cache functionality
- access to external services, such as grid resources, DICOM servers and database engines.
- access to internal tools for processing basic functionality over images, such as assembling of images and conversion of formats.
- multi-database service, for storing metadata

[1] Our implementation of the layer 2 is still primitive, so future work is planned in this area.

– service of files localization in a Distributed System (DS).

5.2.1 API Layer 0

A low level API (referred as API0 in this text) has been developed in order to offer message passing capabilities (layer 0) to the components of the *Engine* and also to independent applications (see section 4.4.2) which connect to the system. Those applications must be registered [2] into the engine and they have to use the message passing API0. The API0 offers functions to send and receive messages to and from a DSE.

A message passing system provides primitives for sending and receiving messages. These primitives may by either synchronous or asynchronous or both. A synchronous send will not complete (blocking the sender) until the receiving process has received the message. This allows the sender to know whether the message was received successfully or not. An asynchronous send simply queues the message for transmission without waiting for it to be received. A synchronous receive primitive will wait until there is a message to read whereas an asynchronous receive will return immediately, whether a message is available from the queue or not.

We use IPC mechanisms [88] [89] in the internals of each engine, and network based communications between different engines or between engines and external services. Although an IPC communication is a local communication only (share memory based), it has a high speed behavior which is interesting for developing high performance operations [115] [114] between components that exchange only local messages; indeed, the engine components interact always in the same computer, even if they have network interfaces to communicate with **other** external components. Thus, we consider that using PVM/MPI at this level is unnecessarily costly (heavy) for implementing local communications, even if other implementation can be developed on them. In the experimentation chapter we present some experiments showing that our API, IPC based, is as efficient as these tools, or in some cases better.

The API0 is based on the Inter Process Communications (IPC) mechanisms of Linux. It hides the complexity of the message passing mechanisms, makes transparent the IPC calls to the operating system and standardizes the message exchange.

For example [3], two functions for sending and receiving a message asynchronously into a standard list of arguments (*msg_ argv* and *msg_ argc*) or into a text (*buffer*), in a microseconds timeout interval, are the following :

```
retcode=DSEMclient_ASYNCSnd(mach_from, mach_to,
&op_code, &msg_argc, msg_argv, reqRspFLAG, &my_sec_pid_ID_back);
```

In this function, a message (argc, argv) is sent from a process (mach_from) to another process (mach_to), and it is specified if the sender waits for a response or not. An important feature of this call function is the assignation of IDs for the target

[2] To be registered into a DM[2] engine means that this engine was configured to accept connections from a specific external application, and performs a greeting process with it before starting-up.

[3] The whole API0 is available on Internet ; see the electronic link at section 9.7

processes (mach_to) ; this means that somebody (the engine) knows about the valid processes (the ones which were registered), and can also resolve (the Message Passing Kernel, below-described) instances of those processes. Thus, this *send function* allows one to identify not only a specific process but a group of processes and to deliver the data to the best target process (the one supposed to be not busy) into this target group of processes. This group of processes are drivers (below-described), which are multi-process implemented. In other words, a driver will use this send routine to deliver a message to another driver which is a multiprocess entity, but, the actual [4] target process is solved by the Message Passing Kernel (MPK).

The symetric receive function is quite similar :

```
retcode=DSEMclient_ASYNCRcv(machine,
     &op_code, &msg_argc, msg_argv, DSE_API_TIMEOUT_USEC,
     &msg_fm, &snd_response, my_sec_pid_ID);
```

In this function an operation code (*op_code*) is sent to the process (*machine*) which is waiting for the message (argc, argv), and it is also informed whether a response is expected by the caller or not (*snd_response*). Similarly, here, the receiving process is selected by the Message Passing Kernel, if it is part of an entity of a higher level (a driver).

These libraries allow one to implement additional functions such as cache, security, files transfer and encryption, database accesses, image tools, etc, which can be easily interfaced, and designed as independent modules for easing software development.

The main API0 functions are :

- DSEMclient_SyncSnd : Sends synchronous messages.
- DSEMclient_ASYNCSnd : Sends asynchronous messages.
- DSEMclient_report_error_byQuee : Sends a response error by using the sending functions.
- DSEMclient_SyncRcv : Receives synchronous messages.
- DSEMclient_ASYNCRcv : Receives asynchronous messages.
- edit_xml_msg : Edits a XML message in order to include useful information for routing of asynchronous messages.
- DSEM_wait_for_N_messages : Waits for a number of response messages, after broadcasting a message.
- DSEM_get_my_machine_number : Gets the internal code for the actual process. This code is the one of the entity of higher level for which the process owns to (the driver).

The whole API0 description is available on the internet ; the electronic link is referenced in the Annexe 9.7.

5.2.2 The Message Passing Kernel (MPK)

The key concept around the level 0 is the collaboration between local processes using an intermediary which is called the *message passing kernel (MPK, defined in*

[4] This is the big difference between an API0 send function and a share memory function of the operating system in Unix/Linux. However, the API0, in its internals, uses share memory functions.

section 4.3.2), and based on inter process communication (IPC) mechanisms. The MPK is a set of processes in charge of providing the routing of messages from one process to another process (see figure 5.1). Our **implementation** of the MPK has a multi-process design, which allows high performance and minimizes the queue of messages to be processed.

The figure 5.1, similarly as the one in last chapter (fig 4.2), shows the network side at top and bottom, but implemented over TCP/IP. The local side (IPC) is shown at sides left and right (of the figure), and has been implemented by using the API0. Processes of different types (see section 4.3.2) are still represented as circles, and inside the square there is a multi-process implementation of the MPK, which also has IPC communication capabilities. In our figure, two processes [5] of the MPK are represented by two texts with the MPK legend, and we can also see that a delivered message to the MPK is processed by the instance with the lowest load. That is why the arrows are directed to different copies of the MPK ; then, the messages are forwarded (arrows going out) to other processes.

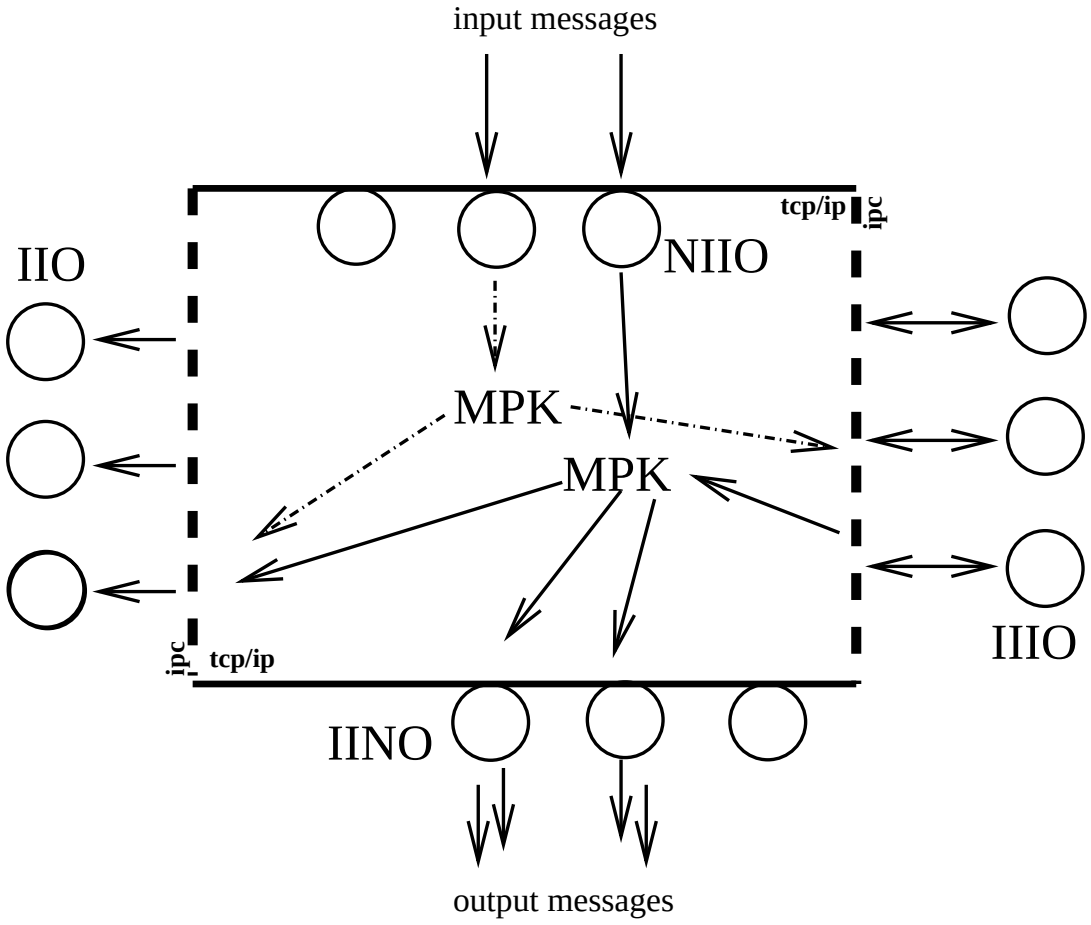


FIG. 5.1 – The Message Passing Kernel

The MPK uses message queues for receiving and sending messages, and never

[5] As noted before, multithreads could also be used for implementing the MPK. In this first prototype we chose develop a multi-process kernel for simplicity reasons.

modifies a message. Because higher level entities such as the drivers are multi process instanced, the MPK deals with tables and criterions for choosing the best target process (i.e. the one with best possibilities) to rapidly process the message which it is delivering.

A typical use of the MPK is described in figure 5.2. It shows how processes are routed by the MPK from a source process to a target process. These messages are delivered by processes using the communication API0 (synchronous or asynchronous functions) and they are also received by other processes which use the same API0; in between, messages are routed by the MPK. For example, a client process IIIO (part of a QUD) issues a message to a Request Driver RQD (composed of processes of type IINO), but the message goes through the MPK. In the figure 5.2 there are defined two processes of the MPK; this is useful for improving performance because messages delivered at the same time can be processed simultaneously by different copies of the MPK [6]. Because the RQD are also multi-processes, the MPK chooses the best RQD process (IINO) to process the message. The incoming messages to the QUD can be delivered by a client machine, and the RQD can issue messages to a server machine. In a DM^2 engine, the same situation occurs when a Service Daemon (SDA) accepts a query (*e.g.,* to retrieve an image) from an external client machine (*e.g.,* a grid), and delivers a request to an external service (*e.g.,* a DICOM server in a hospital), which is managed by a DICOM Request Driver (RQD). The MPK routes the messages in the internals of the engine, and helps in choosing the best processes to process to the messages in each one of the entities (QUD, RQD)

5.2.3 Dispatcher

The *Dispatcher* is a program used to start-up the engine. It reads a set of environmental variables, which previously were registered in a configuration file and exported from there.

The configuration file represents the configuration of an engine in each of the different sites of the D.S. (the engine and its configuration can be different on each site), and is configured to have only the drivers which are necessary per site [7].

The implemented prototype uses *BASH SHELL Environment Variables*, so a *source command* must be executed in order to *export* all of these variables. Next, the dispatcher reads the environment variables in order to get the engine configuration.

A configuration file affects environmental variables (by each driver in the engine), as shown below :

```
export machNR_1=1
export machACTIVE_FLAG_1=YES
export machSTART_FLAG_1=YES
export machNAME_1=HOSPITAL_gibbon
export machIN_PORT_1=6901
export machIP_1=195.220.108.25
export machIP_NAME_1=gibbon
```

[6] the number of copies of the MPK is a parameter of the configuration of the engine.

[7] Some real examples are available in chapter 9

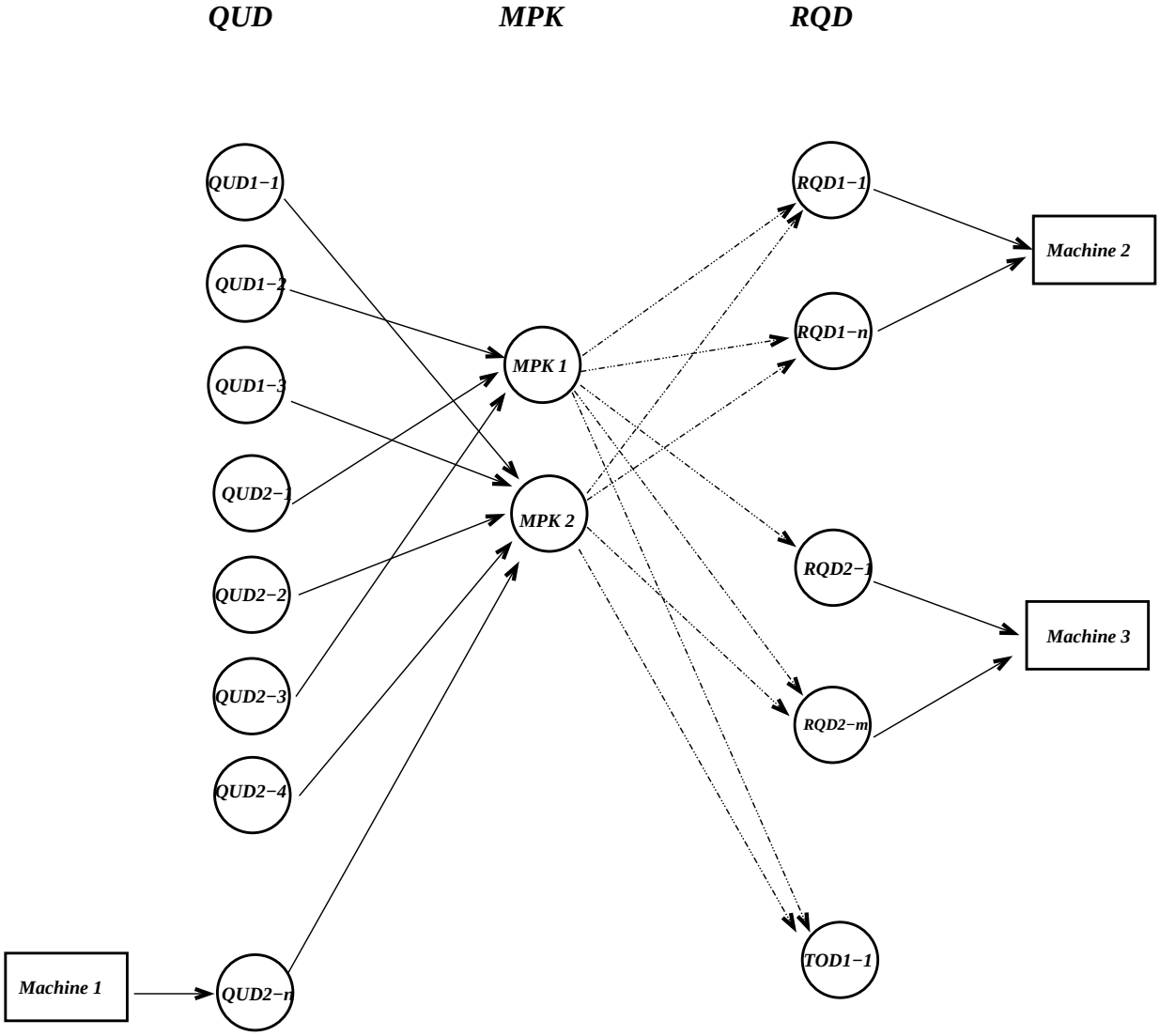


FIG. 5.2 – The Message Passing Kernel Implementation

The left column of circles show 2 multi-process QUD, with 3 and n processes, respectively. In the right column of circles there are 2 multi-process RQD and 1 multi-process TOD. In the center, there is the MPK with two processes. Messages are routed between processes through the MPK. Squares on the left side are the client machines, and squares on the right side are the server machines. Arrows are the messages and their direction inside/outside of the engine.

```
export machPORT_1=7009
export machTYPE_1=DICOM
export machHOST_1=ANY_ONE
export machUSER_1=ANY_ONE
export machPASSWD_1=ANY_ONE
export machDB_1=ANY_ONE
export machNR_SLAVES_1=10
```

In section 4.2 we defined a D.S. as a set of engines joined to a set of machines. The configuration file describes the configuration of each engine, which means that

an engine gets from there the data which it uses in order to auto-configurate itself; however, once configurated, at execution time, what the engine sees are drivers. That is the reason why we talk about drivers while defining the internal components of the engine, because these internal components are made of *DSE*[1] *drivers (QUD, RQD, TKD, TOD)* and services *(SDA and SRD)*. Thus, the configuration file above, means :

- The machine number or ID is 1 and corresponds to a Driver.
- The driver is active. An inactive driver is not considered as being part of the engine.
- The dispatcher must start this machine when starting up the engine. If it is not started, the engine waits for a later greeting procedure of the driver. This is useful for allowing client applications (4.4.2) to connect locally with an engine, or for developing internal plug and play tools and services.
- The name of the driver is HOSPITAL_gibbon. Below it is described that this driver is a DICOM ReQuest Driver type, so here, this name means that the instance of this driver manages one hospital in special : the one which has a host machine named gibbon.
- It uses ports 6901 to 6910 for receiving DICOM images. The dispatcher assumes N copies starting in the port 6901. The number of instances is described below.
- The host name to connect with is gibbon.
- The port where it tries to connect to is 7009. This means that the DICOM service is accessible by this port of the host defined below.
- Its type is a DICOM ReQuest Driver, which means a RQD of type DICOM, or getting connection to a DICOM service.
- The number of processes to be started is 10.

The mechanism of using a configuration file addresses low level issues such as the ports or IP addresses, but it also addresses high level issues such as the number and type of drivers. What is important to notice in this structure is :

- Different engines may have different configuration files, and not different software development. This means that the system is adaptable to the engines types; *e.g.*, as we will see later in this document, a *DM*[2] *server engine* is quite different from a *DM*[2] *hospital engine*, however the software is the same; what changes is the associated drivers and services (different configuration files) - see also chapter 9.
- It enables the definition of different instances of the same driver type, *e.g.*, the defined driver was a RQD for dealing with a DICOM server in one hospital, but we can define as many DICOM RQD as we want, by associating them with different DICOM servers (different hospitals).
- It enables the possibility of *hot configuration*, by allowing external plug and play applications to connect to the engine. This means that a driver or an external application can be connected to the engine while it is running.
- The engine can be configured considering its expected processing charge. This is possible by configuring the number of processes for each instance of the drivers, or the MPK.

The figure 5.3 shows a situation where the application engine is configured to

have one QUery Driver, two Tool Drivers, three TasK Drivers, and two ReQuest Drivers.

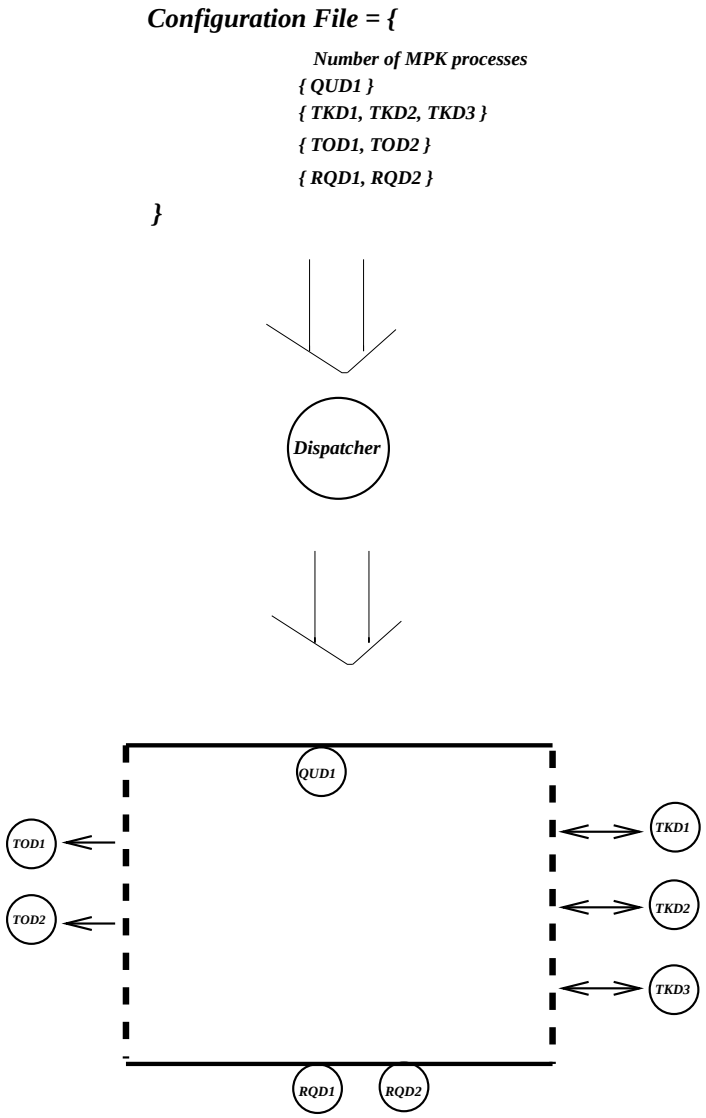


FIG. 5.3 – The dispatcher

The configuration file also exports the variable *mpkNR* in order to define the number of instances of the *message passing kernel (MPK)* to be started.

export mpkNR=2

In this case, we consider that the charge of that engine (in that site) requires two processes for the MPK, but it may vary from 1 to 10.

5.2.4 Monitoring

The monitoring is a tool provided at the lowest level, which plots the size of a queue in long periods of time. Its implementation was done as a IIO process (IPC input only), which is activated from another process.

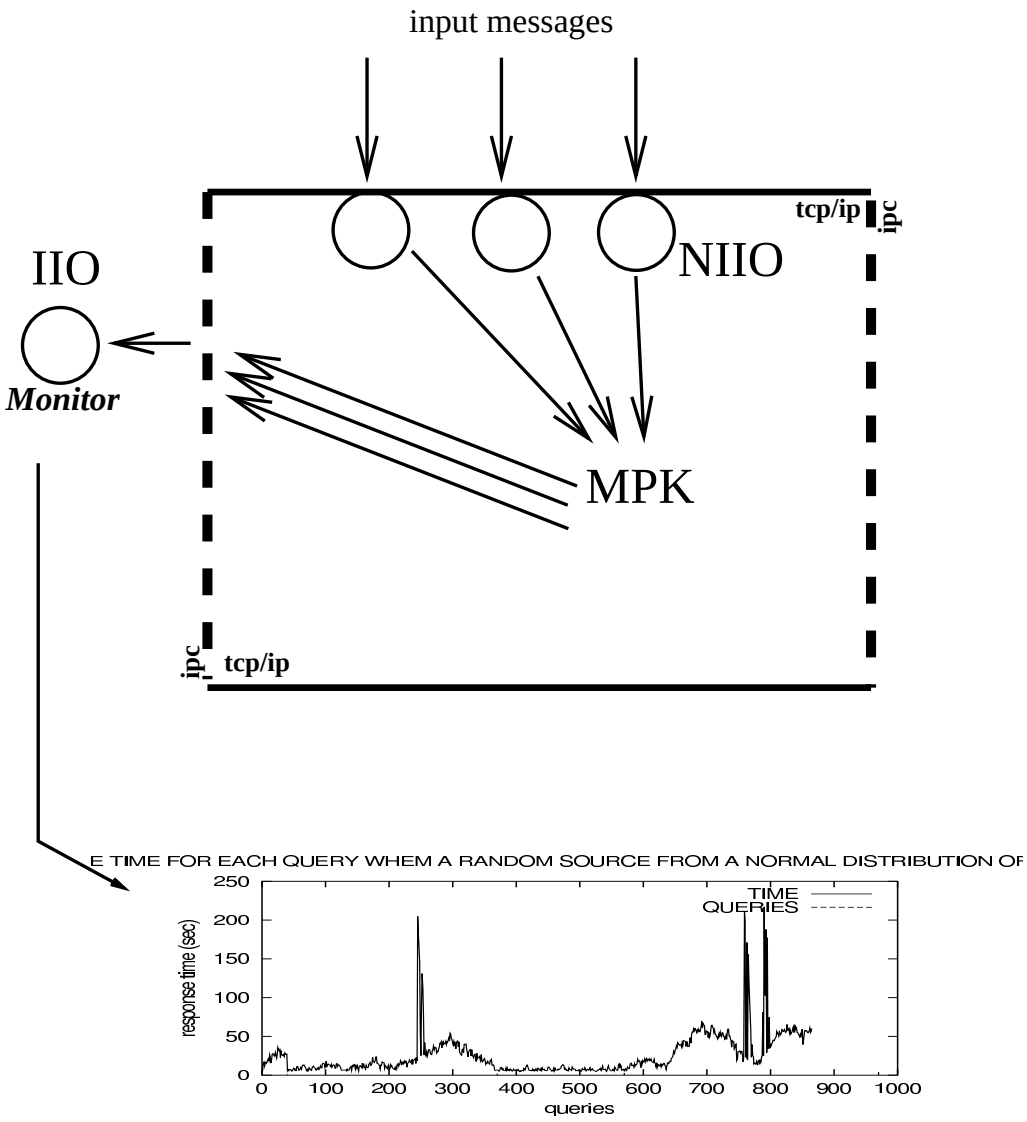


FIG. 5.4 – The monitor

The bottom side of the figure plots the response time in seconds, for each one of the 900 hybrid queries delivered from a random source (normal distribution) - see section 6.2.5

The figure 5.4 shows a situation where a NIIO process is issuing messages to the monitor (IIO process) through the MPK. A real case in our implementation is a QUery Driver which tells the monitor when to start and when to end monitoring. The data to be plotted is obtained by reading the size of the messages queue and the queries queue. This is useful in a DM^2 engine in order to plot the size of the queries queue when stressing the system for performance evaluation.

It corresponds to a basic tool of data plotting, which interfaces standard libraries available in the free community as GNUPLOT [188]. This is a good example of how one can easily integrate this kind of available software, by developing a simple TOol Driver (TOD) which can be interfaced from the whole engine.

5.2.5 Multi Database Structure

DSEM offers the possibility to be connected with several databases at the same time ; this is an implementation usecase of the concept of ReQuest Drivers described in the architecture chapter. For example, an application can be implemented having a TasK Driver for dealing with internal Database Services, however, this TKD can have access to multiple ReQuest Drivers, each one dealing with a specific Database Service (a remote Database Manager System - DBMS). Additionally, each one of these RQD can be multi-instanced. As we will see in the section 5.3.2, this is the way DM^2 has connection with Spitfire (a grid database Service) [138] and MYSQL [196] in different instances (hospitals).

For example, in the configuration file it is possible to include additional environmental variables for dealing with a database service :

```
export machPORT_30=3306
export machTYPE_30=MYSQL_SRV_RQD
export machHOST_30=gibbon.creatis.insa-lyon.fr
export machUSER_30=hDSEM
export machDB_30=dm2
```

In this file [8], we have defined a RQD to connect to a MYSQL database which is running at the *gibbon* host. As usual in MYSQL, the port to connect is the number 3306. At the connection time, it will identify itself to MYSQL as the user *hDSEM*, and opens the database *dm2*. It is clear that by defining another RQD, it is possible to access, *e.g.,* the same database (dm2) in another host, or another database (*e.g.,* security) in the same host (gibbon).

The possibility of connecting with different databases at the same time eases the development of a simple model of distribution. It also enables the facility of accessing different kinds of data, stored in different databases ; *e.g.,* a cache tool or a security tool, would like to have its own data and databases, separated from the applications ones (DM^2), even if they are integrated to an engine, and use the engine for getting access to a database service (see figure 5.10).

5.3 Distributed Medical Data Manager (DM^2)

We have investigated the creation over *DSEM* of a *Distributed Medical Data Manager* unit (abbreviated as DM^2 later in this document) that interfaces the grid middleware. It should provide :

- Reliable and scalable storage for images and metadata produced by medical imagers. This includes connection to the grid file replication mechanism and a metadata location service granting access to distributed medical records. To face the reliability issue in a wide area environment, replication of metadata might be necessary.
- Secure communications, encryption, integrity checking, authentication and a distributed access control mechanism are needed to secure the data [9].

[8] In this case this part of the configuration file makes reference to the driver with code 30

[9] This is the subject of another thesis of our research team [187].

– Synchronization between the medical image data and their associated metadata, due to the fact that they are semantically connected (they should have the same access control patterns, the same processing requirements, etc).

Our *Distributed Medical Data Manager (DM²)* is an implementation of layers 3 and 4 of the DSE architecture, that uses the DSEM. To respond to the requirements and usecases described above (section 2.3) DM² is designed as a complex system involving multiple grid service interfaces and several interacting processes geographically distributed over a heterogeneous environment. It is a grid-aware service as well as an intermediary between the grid and a set of trusted medical sites.

DM² needs to interconnect with existing grid services on the internals of which we have no control [10]. DM² uses external services but also offers services to external applications; with some of them, *e.g.* a grid, there is a cycle in that service offering : for example, a grid offers computing and storing services to DM², and DM² offers medical data access to grid services. It also offers a service of highly semantic structured queries (*hybrid queries*) over medical images for the medical community.

We will refer to *DM² engines* to designate the set of DM² services that we developed in order to avoid confusion with external services. A DM² service is offered by a DM² engine. Each engine is composed of a set of independent drivers which interact by exchanging messages. We designed each *DM² Engine* through the layered *DSE* architecture, described above.

DM² offers these services :

– Queries by content, which allow one to query medical images by looking for special characteristics inside the image itself. In order to resolve these queries, image processing algorithms must been applied.
– Hybrid queries, allow one to query the medical images not only by their content, but also by considering their associated metadata.
– Querying and retrieving sets of DICOM and NON-DICOM medical images, *e.g.,* temporal 3D MRI.
– Metadata management service, for : - Storing and registering new images (result images from experiments). - Managing additional metadata over existing images - Tracking images, for identifying the result images processed from the master images.
– Application of a list of image processing algorithms to an image.
This includes the *Registration* of image processing algorithms in the medical system, and also the *registration* of lists of algorithms to apply to an image at the capture time. These are issues which must yet be developed. Although the functionality of applying the algorithms or a list of them already exists, their registration as a user service is to be developed.

A grid storage interface recognizes files and image processing algorithms which manipulate 3D image files. Therefore, when sets of DICOM slices are registered into an hospital DICOM server, the structure of this data set is interpreted and one or several file IDs are associated to the image files. From the grid point of view, these files will therefore be published and accessible to any grid service through the storage interface. However, the physical image files are not assembled until requested through the storage interface. On demand, the requested image file is assembled in

[10]Indeed these services are not developed by us.

a scratch space [11]. It is then returned to the querier. The image can be replicated to any classical Mass Storage System (MSS) or downloaded to a *worker node* [12] for computation. For efficiency, assembled files are cached in the scratch space for future use. The DM^2 Engine is the interface between the grid and the DICOM servers, or even better, DM^2 offers transparent access to data stored in these servers.

DM^2 also extracts metadata (see figure 5.5) from all DICOM files registered in the DICOM server and stores them in a SQL database to ease the query on metadata. A link between each image, the composing DICOM slices, and the associated metadata are stored in the same database. The metadata structure is designed to be extensible : the user can associate (see section 6.3) any complementary metadata (needed for a medical application) to the image. DM^2 also permits (see also section 6.3) the registration of metadata associated to data files. Indeed, medical metadata are the most critical part of the data as it may contain private and identifying patient information. This process will be clarified in sections 5.3.1, 5.3.2, and 6.3.

DM^2 is able to register and provide a grid interface to data coming from several distributed DICOM servers. As discuted above, it plays a key role in interfacing DICOM servers with the grid (figure 5.5).

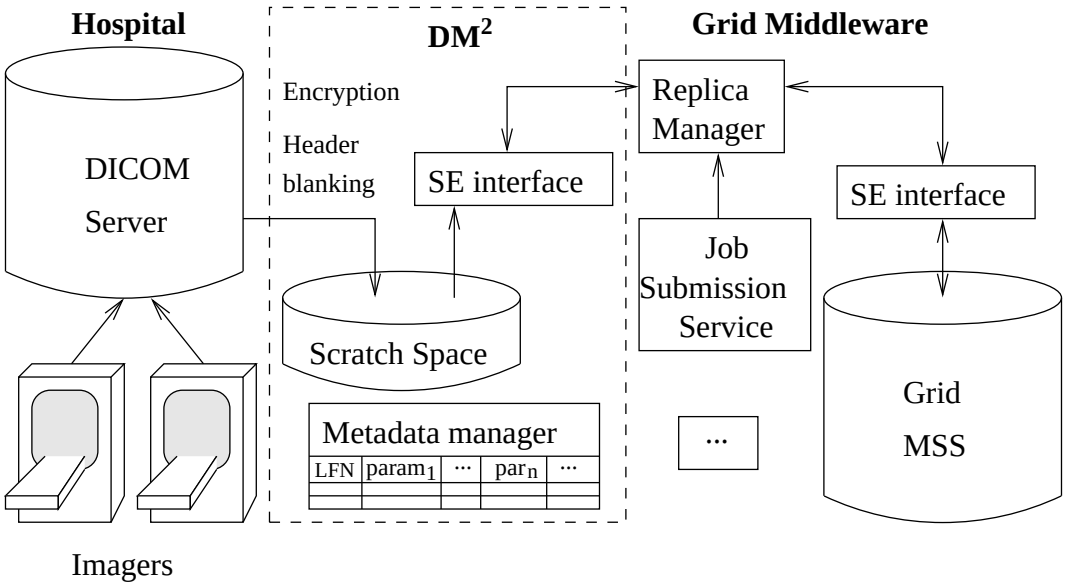


FIG. 5.5 – The DM^2 interface between the medical imagers and the grid

The DM^2 prototype is written as a set of transactions (queries, tasks, requests) and drivers which are separated in different blocks of code called *Packages*. An API (layer 3th or API3) for delivering remote queries to a DM^2 engine is also provided (described below). In section 5.3.1 we describe the most important *DM^2 Queries*, in section 5.3.2 we describe the structure of the code (as packages), and in section 5.3.3 we present the API3.

[11] By querying the DICOM server for the set of DICOM slices composing the image and extracting the image content from these files.

[12] A node in a grid.

5.3.1 DM^2 Queries

We consider as the principal queries those which implement the services of DM^2 [13]. They are :

- DSEM_dm2_getImage_qu : Retrieves all the slices (files of an image) from its storage server, and assemble them in a single file.
- Registers and deletes an image into/from the DM^2 system, which implies to do it also into/from the grid. - DSEM_dm2_regImage_qu
  - DSEM_dm2_delImage_qu

- Basic SQL queries for manipulating metadata - DSEM_jEspy_insert2Table_qu
  - DSEM_jEspy_deleteFromTable_qu
  - DSEM_jEspy_update2Table_qu
  - DSEM_jEspy_selectFromTable_qu

- DSEM_dm2_inqUsingAlgorithm_qu : Applies an existing algorithm to an image, and sends the answer to the user. This query is the base for implementing queries by content and hybrid queries.

A *Query by content* is implemented as the application of a specific list of algorithms to an image, so the Query *DSEM_dm2_inqUsingAlgorithm_qu* becomes of general purpose and usefulness.

For example, consider the code below :

```
   //allocate the struct
1.   inqUsingAlgorithm_msgPtr msgInqUsingAlgorithm = NULL;
2.   msgInqUsingAlgorithm = (inqUsingAlgorithm_msgPtr)
malloc(sizeof(inqUsingAlgorithm_msg));

...

3. retcode=parse_inqUsingAlgorithm_xml(msgInqUsingAlgorithm);

4. if ((library_error=inqUsingAlgorithm_qu(msgInqUsingAlgorithm,
&setofResultsGET))==NO){
         hD_trace("%s going to process a LIST OF ALGORITHMS \n");
}else{
    hD_logerror("%s error processing a DM2 service or a
grid service\n");
}
```

At line 3 a XML [26] [189] message (coming from a remote client machine) is parsed in order to fulfill the *msgInqUsingAlgorithm* structure with the set of algorithms

[13] There are also queries which implement low level functions, but they are not described in this document (see 9.7); *e.g.,* queries for manipulating files instead of images. Similarly we will not describe tasks and requests, because we consider this information as part of a technical document. The suffix **qu** makes reference to the implementation of **queries**

which are going to be applied to an image (its code is also in the structure). The line 4 applies these algorithms to the image.

At the moment, our implementation applies a serial list of algorithms to one image (figure 5.6-ii) ; the figure 5.6-i illustrates a concurrent and synchronized application of the algorithms, but this is work to be done in the future.

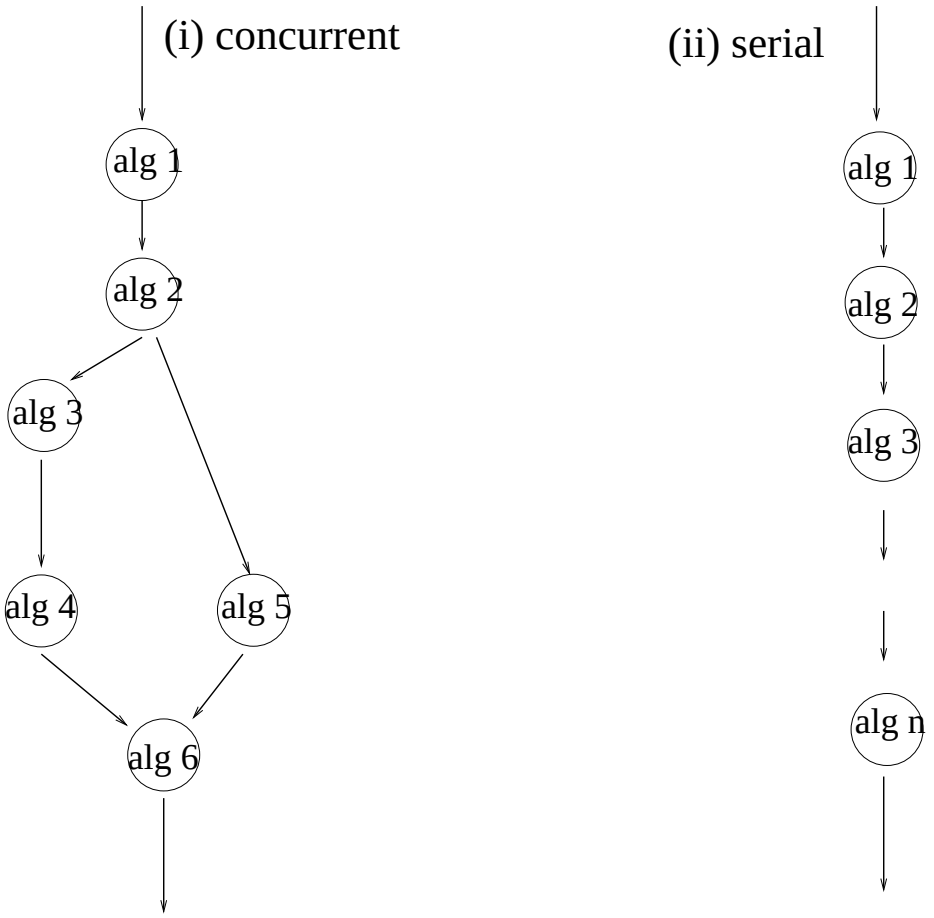


FIG. 5.6 – Application of a list of algorithms to an image
(i) Concurrent Application, (ii) Application in serial

A *Hybrid Query* can be implemented as the combination of a *Metadata Query* and the application of an algorithm (or a list of algorithms) to the image. The section 5.3.3 illustrates this possibility.

Illustration of DM^2 internal interaction

Starting from the detailed usecase described in section 2.3, we sketch the use of the principal *DM*^{2} queries (see figures 5.7 and 5.8).

[26] We use XML because it eases the transmission and sharing of data. It encloses or encapsulates information in order to pass it between different computing systems. The wide use of XML eases implementation, and interoperability.

We have also considered that grid middleware, such as the DataGrid one, uses XML for interfacing other systems.

First, the cardiologist enters a query (*e.g.* to find the MRI recently acquired for a patient in this hospital) through a DM^2 user interface. The user's authorization to access to the data is checked by the security *TOD* (step 1 of figure 5.7-i). The DM^2 engine sends the request (by using the API3 described below) to the metadata interface through the *metadata RQD* and *TKD*. Then, the patient file logical identifier and its associated parameters (imaging modality, region of interest, dynamic sequence, MR acquisition parameters, etc) are returned to the user interface.

Now, the cardiologist enters a request to find all images (see step 3 of figure 5.7-ii) comparable to the image of interest (same body region, same acquisition modality, etc) and for which a medical diagnosis is known. The DM^2 layer 2 should be used here to distribute the requests to all hospitals with metadata services. In the current implementation one single metadata service is queried through the *metadata RQD* and *TKD* again. The logical identifier of all images matching the patient source file parameters are returned.

A request is then made for the computation of similarity measures [28, 27] between the patient image and each image resulting from the query (see step 4 of figure 5.7-ii). The job submission service of the grid middleware is used to distribute computations over available working nodes. For each job started, the grid replica manager triggers a replication of the input files to process onto the grid computation nodes. If the requested files are not registered into the grid (as a replica), they are requested by the grid to the DM^2 engine. The DM^2 engine asks the DICOM server, which assembles MR images on the fly into its scratch space and returns images to grid nodes.

Figure 5.8 details the operation; on top, the grid middleware triggers a DM^2 query for getting an image : (1) It first asks for the image to the *cache TOD*. (2) If that image is not available it then accesses the database (*metadata TKD*) to locate the DICOM files from which the image must be assembled. (3) The *cache TOD* is requested again in order to improve the DICOM file latency access (look for files instead of images). (4) Assuming the cache does not contain the requested file, it should be copied from the DICOM server. The DM^2 engine requests the DICOM server through the *DICOM RQD* and retrieves in parallel a set of DICOM slices that are assembled into the scratch space to produce the 3D image requested. (5) The DICOM files are assembled into a 3D image using an *image TOD*. (6) Finally, the image is stored into the cache and returned to the grid - See step 2 in figure 5.7-i.

5.3.2 Packages

We have written DM^2 as a set of basic applications, called *Packages (PCK)*. Those PCK correspond to all the necessary code for implementing a tool or an internal service, as well as offering access to external services.

These *Packages* are developed over the implementation of the middleware layers *(DSEM)*, and also consider the defined architecture *(DSE)*. For this reason, a *Package* is written as a set of Drivers and Transactions (see section 4.4 in chapter 4). As *drivers*, we wrote *QUD, TKD, RQD and TOD*, which represent Tools, SDA, and SDR. As *transactions*, we wrote Queries, Tasks and Requests.

As an example, for accessing a database external service and also offering a

service (to the DM^2 users) of querying and modifying the database metadata, we wrote a QUD and a set of transactions (queries, tasks and requests), as well as one TKD, and two RQD (for accessing two different kinds of databases) - see figure 5.10. This code is grouped and named as *Database Package* and is described below (5.3.2).

The section 5.3.2 describe all the Packages that we have written ; however, there exists a more general PCK, which has a relationship with all the others and is called the *The DM^2 Core Package* (described in section 5.3.2).

The DM^2 Core Package

The main *Package* is the *DM^2 Core Package*. It develops the access to a DM^2 engine and the assignation of tasks and requests to the drivers. For this reason, its code is composed specially of QUery Drivers (QUD) and Queries.

These include :

- A QUery Driver (QUD) for giving access to other DM^2 engines
- A QUery Driver (QUD) for giving access to external machines such as grids or clients using the API3 (section 5.3.3) , *e.g.,* external machines or applications delivering DM^2 messages.
- The set of queries which represent the offered DM^2 services, such as hybrid queries, queries by content, queries to metadata, and access to images.

Each one of the queries is a set of tasks and requests (see chapter 4). In order to resolve each one of these queries, different internal drivers must be accessed, depending of the requested service. For example, querying an image means doing interactions with drivers of the database service, but also with drivers of the file retrieval service (DICOM). Thus, such a query can be implemented as two tasks : localizing the image, and retrieving the image. Because those tasks are implemented in two different packages (database PCK and DICOM PCK), a query such this one, which uses tasks from different packages, is written in the *DM^2 Core Package*.

Our implementation includes with the DM^2 Core Package those queries which interact with different services (*e.g., querying an image above*) . The others are included in the implementation of each Package ; *e.g., query of metadata* does only needs to interact in the internals of the *packages*, so this query is included there.

The figures 5.9 and 5.12b represent *Packages (PCK) Schemas* [25], which describe what is included with the DM^2 Core Package (written source code). For example, figure 5.9 shows that the DM^2 Package has two *QUery Drivers (QUD)* and some *Queries*. Figure 5.12b shows that the same two *QUery Drivers (QUD)* of DM^2 also deal with queries to execute in a *DM^2 Hospital Package* as described below in section 5.3.2.

[25] Package Schema : Similarly to the other figures, the square represents the DM^2 engine, and circles the drivers ; but in this figure they are dotted in order to call the attention in the fact that they represent a different type of schema. The internal polygons represents what is included as **written code** for this Package (**PCK**). If its surface covers the circles, this means that some drivers of that type are included, if a directed triangle is designed, this means that transactions of that type were also included with the source code of the Packages (PCK).

DICOM Package

The *dm2DICOM Package* was developed for dealing with DICOM3 servers. In figure 5.11c the surface [25] representing the *DICOM Package* shows that it is composed of :

- A TasK Driver TKD which provides transparent and concurrent access to a internal DICOM service (SDR)
- A ReQuest Driver which implements session establishment and operations with a DICOM3 server (usually CTN or DCMTK).

DM^2 uses this Package for doing *pull* actions of DICOM images from a Hospital. Each instance of the RQD is connected to a DICOM Server (CTN) which represents a hospital, and each one of these instances is started as a multi-process driver, which allows the parallel transfer of DICOM files from each hospital. In this way, a DM^2 engine can manage multiple hospitals by starting multiple instances of the DICOM RQD.

The RQD implements the Query Retrieve Service Class (QRSC) defined in the DICOM3 standard [49], by using the DICOM Tool Kit (DCMTK) [136].

Database Package

The *dm2DB Package* was developed for dealing with the Database Service. In figure 5.11b the surface [25] representing the *Database Package* shows that it is composed of :

- A TasK Driver TKD which provides transparent and concurrent access to a internal Database Service (SDR)
- A ReQuest Driver which implements session establishment and operations with a MYSQL Database Management System (DBMS).
- A ReQuest Driver which implements session establishment and operations with a Spitfire Server [14].
- Basic Queries for doing SQL operations (select, insert, update, delete) over the metadata database in the server engine side.

At the moment our prototype is working with the MYSQL RQD. The Spitfire RQD was also tested but there were problems of stability with the C++ API which was provided by the DataGrid Work Package 2 team. Some work must be done in order to adapt it for LGC2 (EGEE).

The TKD enables the queries to access transparently and concurrently different databases, or also the same database with different SQL operations. The same TKD is valuable for the two RQDs, which means that development of the TKD is not dependent on the implementation details of the MYSQL RQD or Spitfire RQD.

The joined action of the TKD and RQD enables the implementation of a *Multi Database Structure*. DM^2 uses DSEM as middleware, so it has the possibility to be connected with several databases at the same time ; *e.g.*, a DM^2 server can have simultaneous access to its central database and to remote DM^2 client databases (hospitals). For example, a DM^2 server managing N clients (hospitals) can be configured to be connected at the same time to its central database and to N DM^2 client data-

[14] DataGrid middleware for accessing SQL Database Management Systems (DBMS)

bases (hospitals) . This means to be connected simultaneously to N + 1 databases - see figure 5.10. Additionally, it is possible to configure additional databases for managing security or cache issues. This will be useful when the security and cache systems have been integrated with DM^2.

The definition of the server and client databases for DM^2 are described in annexes 9.5 and 9.6.

Grid Package

The *dm2Grid Package* was developed for dealing with the Grid Services of resources sharing (storing and computing).

In figure 5.11a the surface [25] representing the *Grid Package* shows that it is composed of :

- A TasK Driver TKD which provides transparent and concurrent access to a internal Grid Service (SDR)
- A ReQuest Driver which implements session establishment and grid operations with the MicroGrid software [131].
- A ReQuest Driver which simulates session establishment and grid operations with the DataGrid middleware.

The Grid TKD can access each one of the RQDs (DataGrid or MicroGrid), but only one can be configured and started at a time. In section 5.3.2 we will see that the DataGrid RQD is only a simulated interface, so the operational one is the MicroGrid RQD.

The TKD also offers the possibility of managing the parallelism of tasks with the Grid Service, and transparency ; *e.g.,* in order to integrate another (external) Grid Service, only a new RQD must be developed (not the TKD).

DataGrid RQD

A grid middleware, such as the EDG (European Data Grid) middleware [126], proposes a standard storage interface to the underlying Mass Storage Systems (MSS). Through this interface, the middleware can access files located on distributed and heterogeneous storage pools. Grid-enabled files are handled by a *Replica Manager* (RM) : to ensure fault tolerance and to provide a high data accessibility service, files are registered into the RM and may be replicated by the middleware in several identical instances. The first file registered into the RM is a *master file*. Other instances are *replicas*. When a file is needed, the grid middleware will automatically choose which replica should be used for optimizing performances. Having multiple instances of a file also increases its availability since connection errors are likely to happen in a wide scale distributed system. To solve coherency problems, replicas are accessible in read only mode and modifying a master file invalidates all its replicas. To ease files manipulation, grid wide *Logical File Names* (LFN) are used to identify each logical data (*i.e.* a master and all its replicas).

For each new DICOM image or set of DICOM images (depending on the semantic of the DICOM series) produced by an imager, a LFN is created and registered into the RM. The DICOM files thus become, from the grid side, a master file. There is not necessarily a physical file instance behind this LFN but rather a virtual file

made up of a set of DICOM files, that can be reconstructed on the fly by DM^2 if a request for this LFN comes in. For efficiency reasons, assembled files are cached in a scratch space before being sent outside. DM^2 also stores metadata and establishes a link between an LFN and its patient- or image-related metadata.

The DM^2 storage interface ensures data security by anonymizing on the fly images that are sent to the grid. Replicas of a medical image may exist on any grid storage node, given that encryption forbids data access without decryption keys. These keys are stored with the patient-related metadata on trusted sites only [48]. In order to ensure data integrity, the grid storage interface does not allow the master files stored on the DICOM server to be deleted.

The grid enables the DICOM server with a storage interface that makes it visible as any MSS. However, DM^2 acts as a read-only Mass Storage System as it does not allow external grid data to be stored on the sites it controls : new medical images are registered internally when produced on the medical imagers and DICOM servers are not intended to store any other kind of data.

These functionalities were implemented as a DataGrid ReQuest Driver (RQD) for issuing messages to the DataGrid, and as a QUery Driver (QUD) for accepting queries from the DataGrid. The messages interface as was defined with the DataGrid Work Package 5 team is shown in the chapter 9. It was tested in simulation mode but, unfortunately, the *Biomedical Applications (Work Package 10 - WP10)* were not of the highest priority for the DataGrid project and an integration with the DataGrid middleware was not possible. At the moment, the DataGrid project [126] was replaced by the EGEE project [143] and the new middleware (LGC2) does not consider the work of the Work Package 5 team. Therefore, additional work must be done in order to integrate DM^2 with LGC2.

MicroGrid RQD

The MicroGrid software [131] is a light-weight grid development intending to provide basic grid functionalities for users through an API and a command line interface.

It provides multiple computation and storage resources management and can be easily installed on a cluster in order to prototype a real grid environment.

We implemented a RQD using the C++ API in order to establish a session with a MicroGrid, enabling access to file registration and job submission services. The DM^2 engine uses it for registering images and submitting algorithm computations on the images.

The power and versatility of the architecture enables DM^2 to use different grids as a source of computing resources. The Grid TKD can deliver requests to a internal Grid Service, without worrying about the specific implementation of Grid : DataGrid, MicroGrid, or LGC2, in the future.

Hospital Client Package

The figure 5.12a shows the Package for a *DM^2 Client Engine* at a Hospital. This Package is composed [25] of :

– A QUery Driver for synchronously accessing a DM^2 Server Engine

- A QUery Driver for asynchronously accessing a DM^2 Server Engine
- A set of Queries for registering and deleting sequences of images and slices (DICOM and NON-DICOM).
- A command line utility for testing the queries.

A hospital can be configured to use a synchronous or an asynchronous RQD. The function of these RQD is to have communication with a DM^2 Server Engine in order to register the images into the system, and to transfer the associated metadata for each image.

Each hospital manages different Magnetic Resonance Image (MRI) devices for image capture (see section 6.3.2); thus, the DM^2 Client Engine must deal with different and concurrent scanners as well as with multiple sequences of images in order to register them into the system. Each image can be made up of one or several DICOM files representing portions of the imaged body. The underlined problems are :

- to intercept the capture event [15] (of the image) and to process this event into the DM^2 system, which means to start the necessary processes at the DM^2 Client and Server Engines.
- to extract its useful metadata and to build relationships of a high level between the set of DICOM files of an image ID. This allows one to identify, *e.g.,* a temporal sequence of slices (of a 3D image) under a single image identifier (ID).
- to update the database in the DM^2 Server Engine.
- to compute special characteristics of the image (image processing) and to register it as special metadata.

The Hospital Client Package provides the software components to analyze the sequences of images in order to determine when a sequence finishes or another one starts. This allows one to decompose the sequences of images in all its associated slices, and to register these relationships into the database. It also provides the queries for starting the computation (in an external computing resource, as a grid) of characteristics in the image.

This PCK includes also the necessary components for intercepting the capture event as described in section 6.3.2 in chapter 6.

Tool Drivers Packages

We have developed two TOol Drivers (TOD), one for assuring basic cache operations, and another for dealing with images operations -see figure 5.13. Another group of our team [187] also implemented a prototype of a security TOD. The figure 5.14 shows the integration of the TOol Drivers which are being developed by our research team [187] as independent projects.

[15] By *capture event* we refer to the action of pushing a DICOM image into the DICOM Server, after it is scanned. As shown in section 6.3, these images go into a server as CTN or a PACS system, not to a DM^2.

Cache Package

The system performance is sensitive to the quality of the link to hospitals where the DICOM files are stored. Our team researches in order to develop cache techniques which could be useful for the DM^2 system, in order to improve the latency of image access.

We have one vision of such a cache system as a TOol Driver (TOD), but its final integration with DSEM/DM^2 is yet to be done in the future; however, we have implemented all of the interfaces with a basic cache which at the moment can be used as a TOD. The integration of the new cache system, will only require to replace the TOol Driver.

That vision proposes several levels of cache in order to improve the latency of accesses :

- First, there is a *request caching* layer. The goal is to cache requests and results in order to have a high probability of finding precomputed responses [46] to incoming queries to the system.
- Second, the complete image is expected to be residing in the cache area (*image caching* layer).
- If not, the *file caching* layer works in order to find some DICOM files (image slices) in the cache area.
- Finally, having no other option, DICOM files are transfered in parallel from the hospital, then registered into the cache. The image is assembled and also registered into the cache.

The cache must assure these functionalities :

- cache responses instead of only files and data [16].
- cache set of files (images and sequences of images).
- check the existence of files in cache.
- register files in the cache.

The cache package which is integrated is only composed of a TOol Driver, as it is a basic cache only.

Images Package

When a sequence of images is required, DM^2 goes to the hospital where the image is stored, and makes a DICOM pull of the images to the DM^2 server. This sequence of images is stored as a set of DICOM files in the hospital, so the set of DICOM files is transferred to the DM^2 server. We have developed a TOol Driver for assembling on the fly these files (images) into an unique file [17] before sending it to the grid or Client.

Additional functionality, such as format conversion or header manipulations of DICOM images can be implemented here.

The images package is composed of a single TOol Driver.

[16] This functionality is not implemented, but it will be included in the future cache

[17] Multiple formats are supported : JPG, GIF, INR.

Security TOD

Data management and replication mechanisms [7] proposed by grid middlewares mainly deal with flat files. Data access control is handled at a file level. In the DataGrid (and EGEE) project for instance, user authentication relies on the asymmetric key-based Globus Grid Security Infrastructure layer (GSI) [42]. This infrastructure does not take into consideration metadata and can not address patient record distribution.

Preserving patients privacy is a major concern for medical data processing systems. The distribution of data over a grid makes data control much more difficult than on closed systems. Data on grids may be replicated but all storage sites are not accredited to receive medical data. Therefore, their administrators should not have read access to the data content. Some identifying metadata are not accessible to non accredited users as well. Achieving a high security level is mandatory but security is always a trade off between inconvenience for the users and the desired level of protection. In order to convince users (physicians and later patients) to use grids for their data storage and processing needs, many functionalities need to be provided such as :

- Reliable authentication of users.
- Secure transfer of data from one grid element to another.
- Secure storage of data on a grid element.
- Access control for resources such as data, storage space or computing power.
- Anonymization of medical records to make them available for research.
- Tamper-proof logging of operations performed on medical files.
- Robustness against denial-of-service attacks
- Traceability

The features that should protect data [91] while it is being processed on a grid, are access control and anonymization. Users need to trust the servers on which their data are going to be processed. To our knowledge no systems exist for data processing on untrusted resources.

Our team [187] is doing research in addressing all these requirements. A TOol Driver prototype was developed and tested with DM^2, however, it is not detailed here because it is part of another thesis. That TOD uses the API0 in order to get connection with DM^2, and once finished it will address : (i) authentication, (ii) secure transfer, (iii) secure storage of data, (iv) authorization and access control, (v) anonymization, (vi) traceability.

The security is not a package, because its code is not integrated with DM^2, but the system which is being developed can be integrated as a TOD of the DSEM system.

5.3.3 API Layer 3

This API (called API3) delivers queries to a DM^2 server engine. Using it, an application can be coded to became a DM^2 client. Usually, on top of this API3 there is a command line program or a GUI.

The API3 delivers remote XML [26] [189] messages to a DM^2, and it allows these operations or queries :

- registers a slice or a sequence of DICOM images.
- registers a slice or a sequence of NON DICOM images.
- deletes a slice or a sequence of images.
- SQL queries (select, delete, update, insert) over a DM^2 table.
- pushes an image or slice to DM^2.
- pulls an image or slice from DM^2.
- moves an image to the grid.
- applies a list of algorithms to an image.
- associates text and image files to an existing image.

Some code at layer 3

A very simple C program to do the above process, can be written as follows [18] :

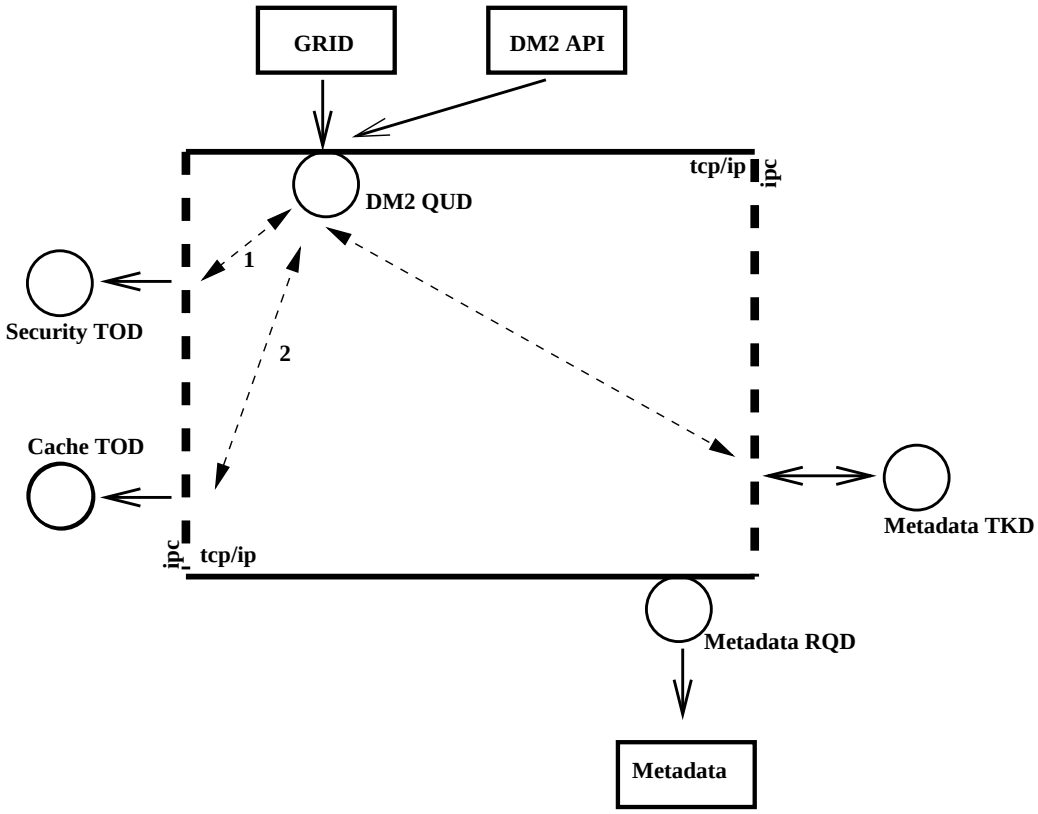
```
1   strcpy(requestToDM2, "WHERE patientID="MR X" AND date="31May2005"");
2   inqdm2image (&RequesttoDM2, &Imageslistforpatient);
3   strcpy(DM2imageid, Selectimagefrom(Imageslistforpatient));
4   DSEMapi_3_dm2_inqUsingAlgorithm_qucl (DM2imageid, &Algoparametersforimage);
5   DSEMapi_3_dm2_inqUsingAlgorithm_qucl (DM2imageid, &ComparablerequesttoDM2,
        &Algoparametersforimage, &Comparableimageslist);
6   DSEMapi_3_dm2_inqUsingAlgorithm_qucl (DM2imageid, &Algoparametersforimage,
        &Comparableimageslist, &Similarimageslist);
7   for (Img=0; Img<N; Img++){
8      Myprocess(Similarimageslist[Img]);
9   }
```

For simplicity of reading, we do not include variable definitions. At line 1 and 2 we build a query which is delivered to the DM^2 system in order to get a list of images for the patient "Mr X" which were acquired on a specific day. One of these images is selected (line 3) considering a user's defined criterion (the function *Selectimagefrom*), parameters of interest for a similarity algorithm are computed (line 4), and then an image database is queried (lines 5 and 6) to obtain similarity measures from the selected image. A list of similar images is returned (line 6) and the user starts his process (lines 7 to 9) to do the image processing tasks.

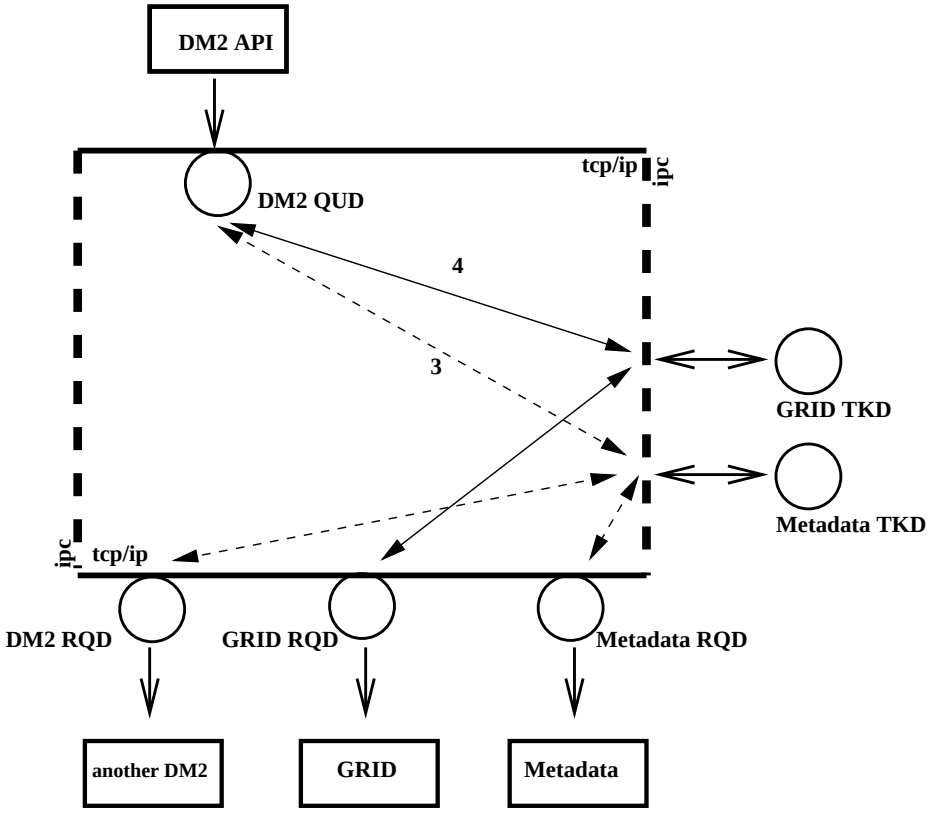
The whole API3's description is available on the Internet ; the electronic link is referenced in the Annexe 9.7.

[18] This code is different from the one presented in section 6.3. In that, this is a use of the API3 (client side) functions, whereas in section 6.3 we use commands which call these API3 functions. Similarly, this is different from the code presented in section 5.3.1, which corresponds to the queries on the server side

–



(i)



(ii)

FIG. 5.7 – DSE[3] usage example : a hybrid query .
(i) 1-Security check 2- Cache query for stored results, (ii) 3- A distributed request is issued to find all images comparable to the image of interest, 4- similarity measure process is issued to the grid. See section 5.3.3)

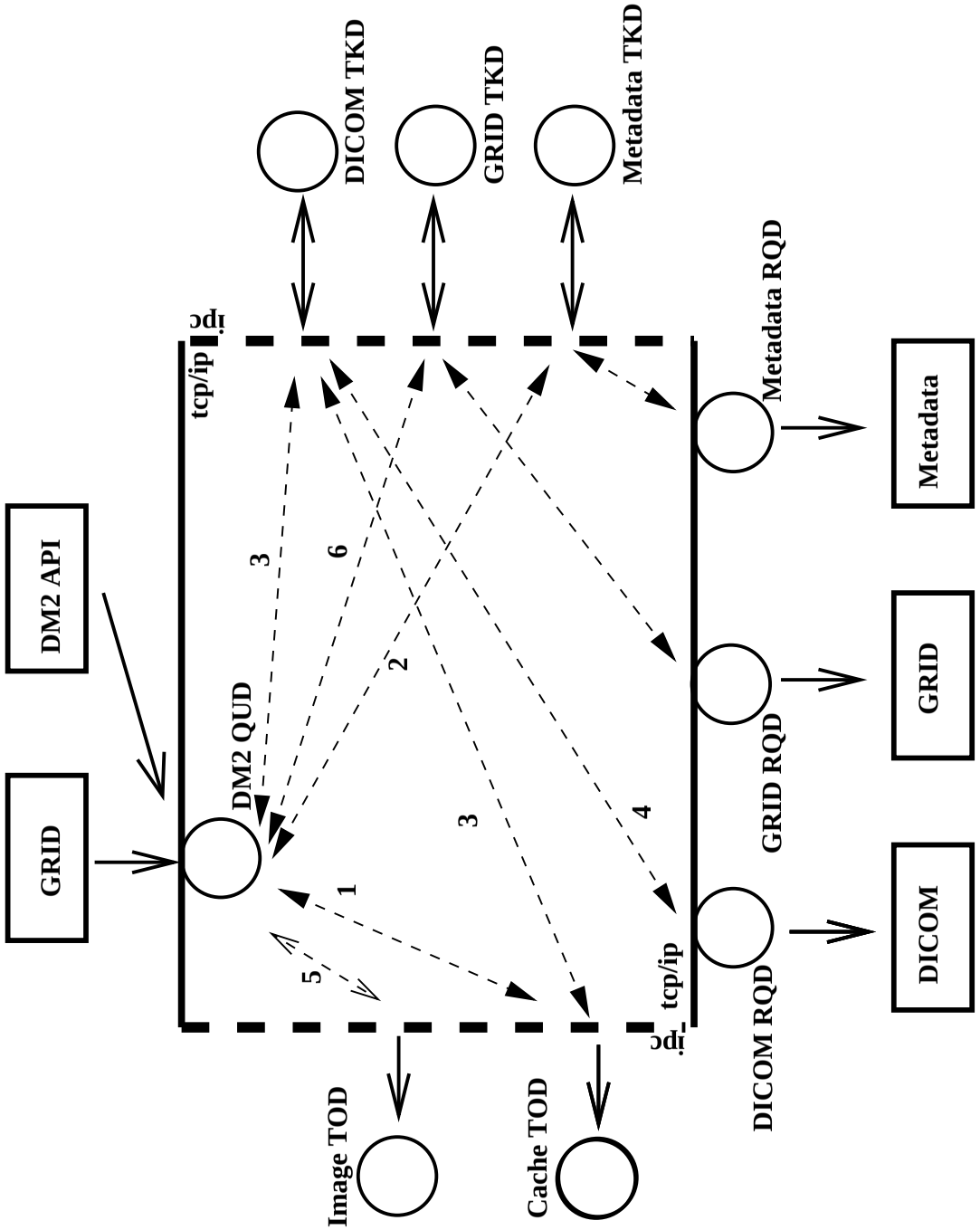


FIG. 5.8 – Usage Example
Retrieving a medical image from the DICOM server (a set of 2D slices composing a temporal 3D image for instance) .

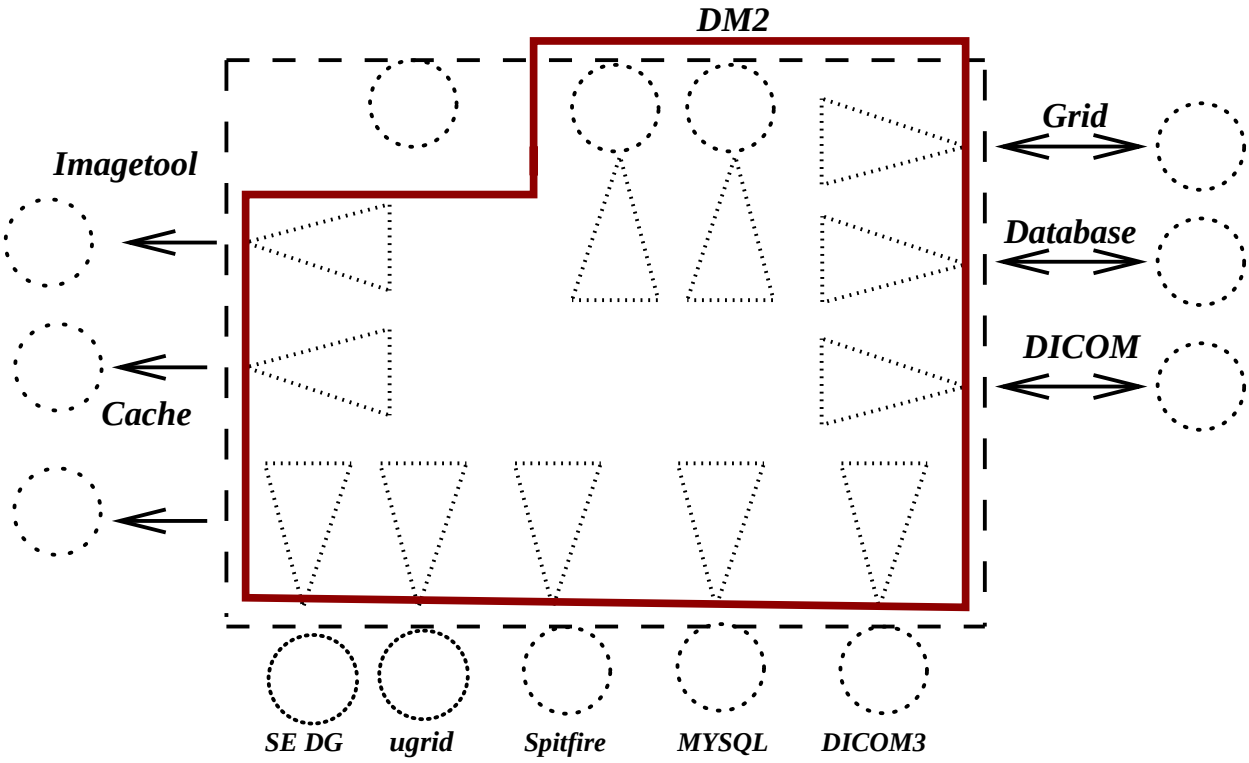


FIG. 5.9 – DM^2 Core Package

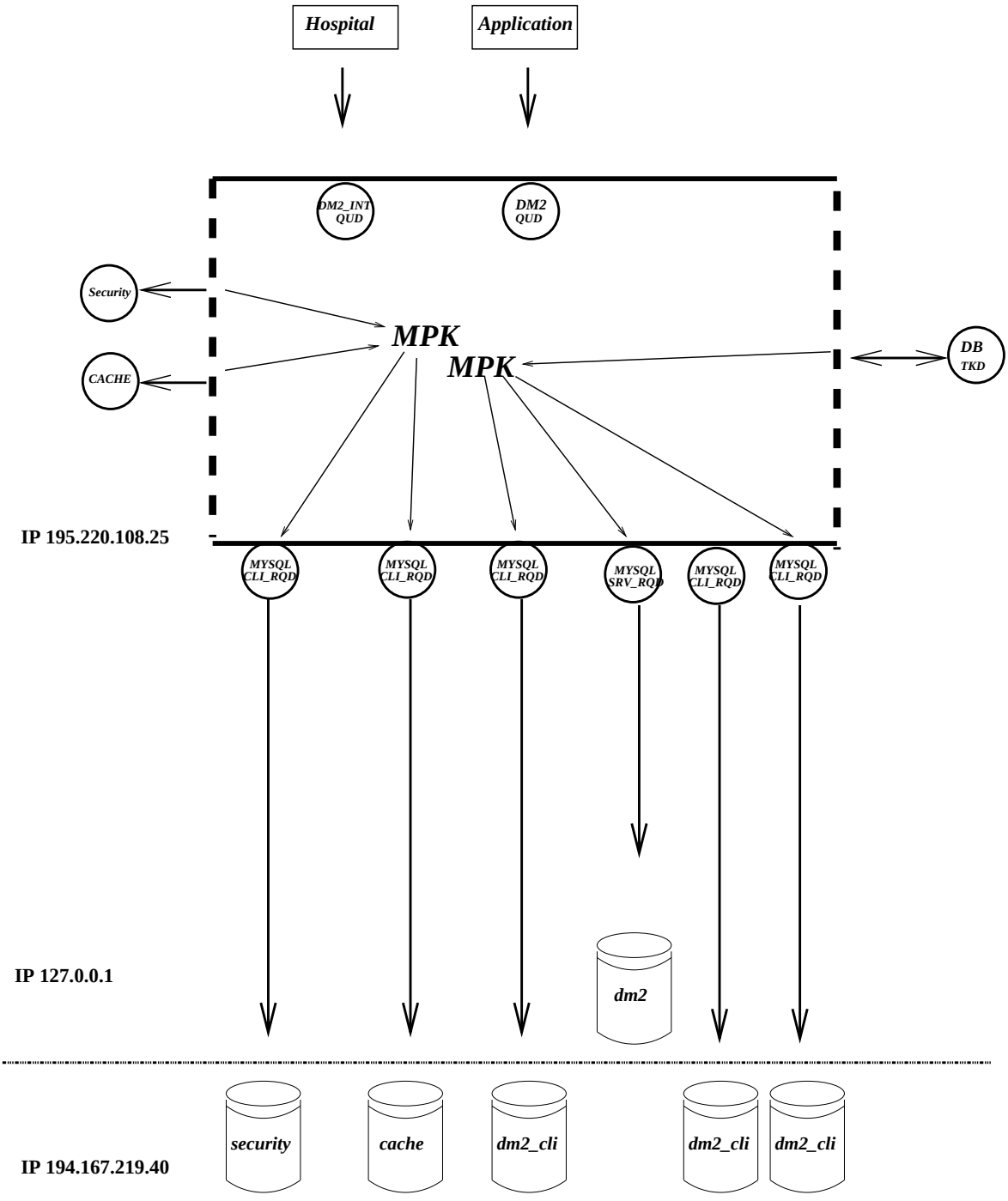


FIG. 5.10 – Multi database structure

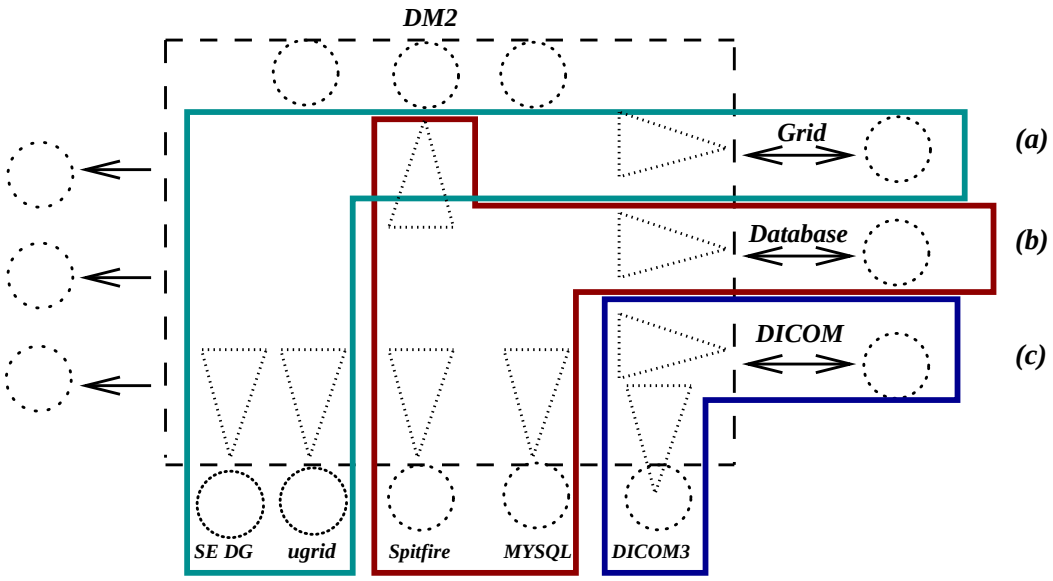


FIG. 5.11 – Packages (PCK)
(a) Grid PCK, (b) Database PCK, (c) DICOM PCK

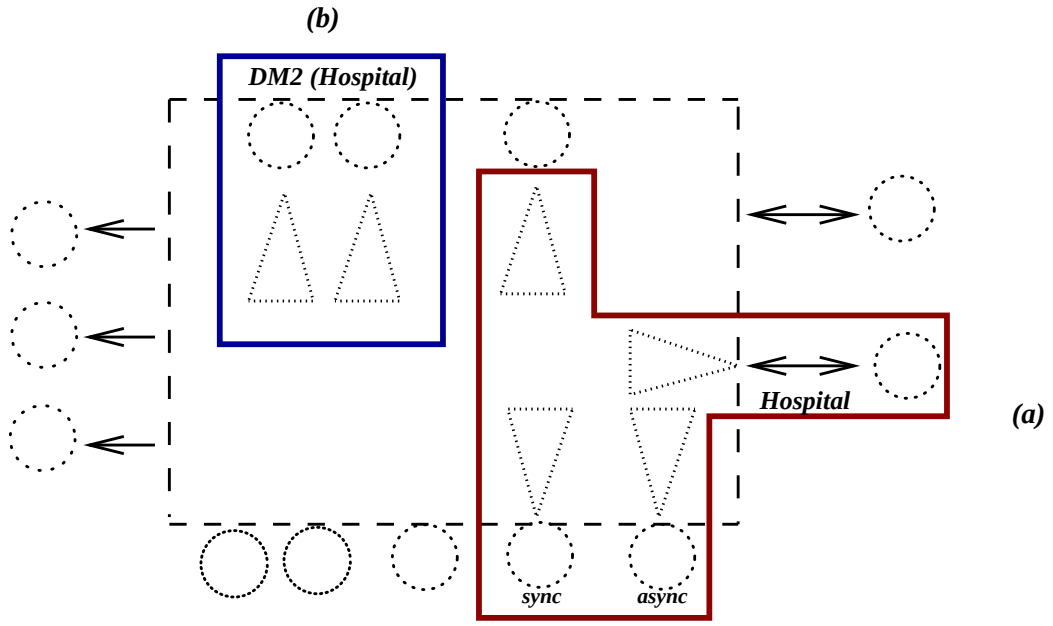


FIG. 5.12 – Package - Hospital
(a) Hospital Client Package, (b) The DM^2 Core Package covers the Hospital

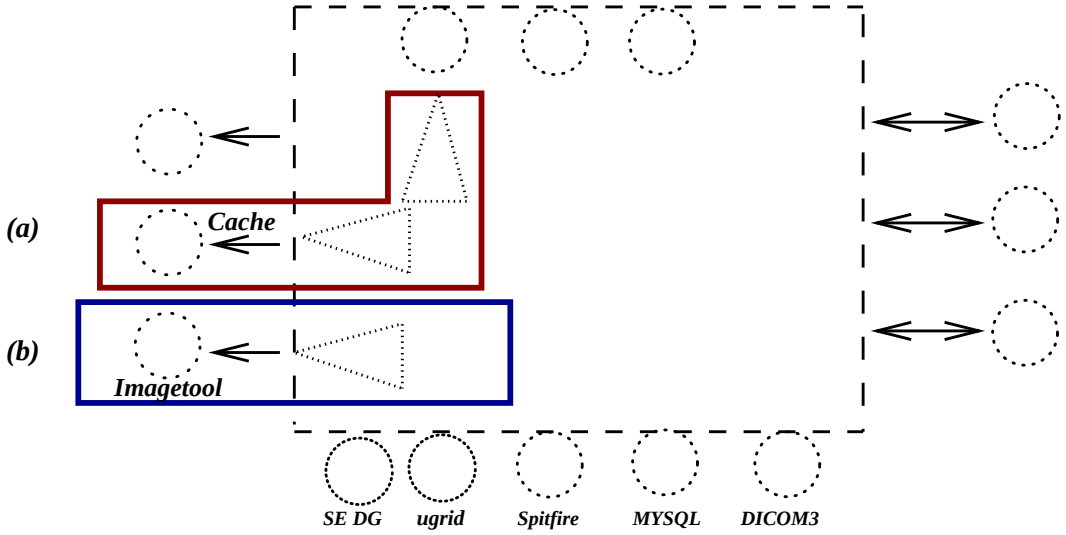


FIG. 5.13 – Tool Drivers
(a) a basic Cache TOD, (b) image TOD

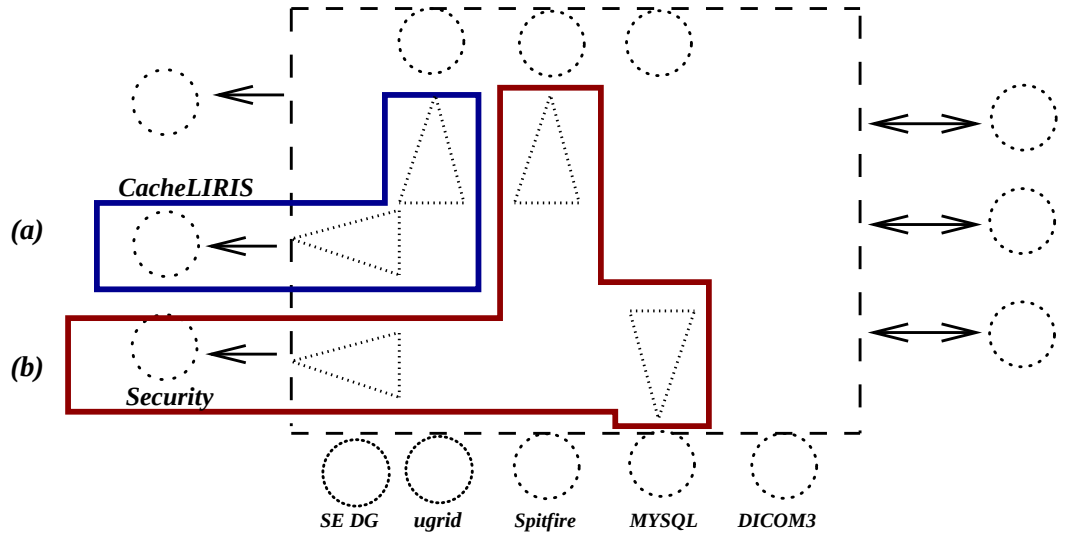


FIG. 5.14 – Tool Drivers Integration
(a) the cache TOD which is being developed, (b) the security TOD which is being developed

Chapitre 6

Experimentations

*"The key is to offer, select and aggregate resources based on individual requirements.", **Rajkumar Buyya**, University of Melbourne, Australia*

*"Users are not concerned with where the data is located as long as they have access to the data", **Carl Kesselman**, University of Southern California, USA*

–

Summary 6

In the first part of this chapter we describe tests of performance of a DM^2 engine, and in the second part a medical application is presented.

At first, we present some experiments designed for analyzing the behavior of a DM^2 engine. We decompose the response time of a query into its main phases, and we recognise the importance of the DICOM phase (access to data). Thus, we do additional experiments to see the behavior of the system in stressing conditions when performing the data access.

We test the system working when it rises its data access capacity, which means when its data access resources become saturated, and also when these are over saturared. We also analyze the system when queries are issued by random intervals of time.

Additionally, we check by experiments to analyze the performance of the message passing layer.

Results shown that the performance of our prototype ($DSEM^0$), for making operations of message passing, is better that PVM. The prototype is also stressed for solving up to 800 DICOM file transfers (pulls) in parallel, and up to 10000 queries arriving with a random pattern of time. All cases shown that the system can deal with the efficient evacuation of the work.

Then, we present a medical application which uses the DM^2 system, and which we test as a prototype implementation. This application is presented from the user point of view. We place the emphasis on the 4th DSE layer (user).

We summarize the whole DM^2 system, and also present the mechanisms of image capture, as they are implemented in the Cardiological Hospital of Lyon. This particular implementation uses a DM^2 Client Engine and two DICOM Servers : CTN and DCMTK.

Finally, we present two medical usecases : Similarity and Segmentation of Cardiac Volumes. These usecases correspond to the application of dedicated image processing algorithms, to medical sequence images. The users deliver queries to a DM^2 Server Engine, and those are resolved by querying metadata, accessing the images, and using Grid computing resources. These queries correspond to hybrid queries and queries by content, and they are very time consuming.

–

The emergence of data-intensive medical applications, require high performance transport and replication of very large data sets among multiple geographically distributed sites. For the overall performance of grid applications, the data access time is as much of a critical component as the computational speed.

A set of DM^2 engines interacting and working together are a kind of virtual machine that unifies data sources (hospitals) with computing and storing resources (Grid). This unification aims at providing more and better services [1] than available in a single DM^2 engine server; thus, the quality of service is related to speed and availability.

The availability of some DM^2 services (hybrid queries) depends mainly of the availability of its computing resources (Grid) and storage resources (Grid and Hospitals); however, its speed, or data access performance, is mainly controlled in the DM^2 engine by using concurrency. Thus, we have designed some experiments to measure the performance of an engine in terms of data access time. These experiments are presented in the first part of this chapter (section 6.2).

In the lowest level, everything relies on the message passing paradigm, therefore we began by testing the performance of that layer (*DSEM*^{0}) in section 6.2.1. Data access is the most time-consuming [2] phase in the solution of a hybrid query, so we have designed four experiments to measure the performance of a DM^2 engine when retrieving an image (set of slices) from its DICOM storage source (hospital) over stressing conditions.

The first experiment analyzes the behavior of such a query, decomposed of phases (section 6.2.2). The second experiment brings the system to its maximal capability (section 6.2.3), the third goes over that thereshold (section 6.2.4), and the last shows a more realistic situation (section 6.2.5) by implementing a random arrival of those queries. Experiments two and three deal with queries arriving in parallel in order to produce bottleneck situations.

In the second part of the chapter (section 6.3), we present a medical application from the user point of view. We define an environment with a DM^2 server engine and eight hospitals (section 6.1), and then we test it by delivering concurrent queries for accessing and retrieving data. We use the query (*getDM2Image*) which localizes, retrieves, assembles and finally copies an image to the Grid.

6.1 Test Environment

We have configured (see section 9.4 in chapter 9 for experimental conditions) a *DM^2 Server Engine* with eight attached hospitals (DM^2 Client Engines), with each hospital running one DICOM server (CTN). The Server Engine was configured to manage up to ten sessions in parallel with each one of the hospitals, so it can transfer in parallel as many slices as defined sessions. This means that our server can deal with pulling up to 80 DICOM slices in parallel (10 slices per hospital).

```
capacity= 8 hospitals * 10 sessions/hospital
```

[1] distributed hybrid queries for the medical case.

[2] Without considering the time of running an algorithm, which must be highly costly, but which depends of the algorithms by themselves, and on the Grid computing performance

```
capacity= 80 sessions (or slices in parallel)
```

Figure 6.1 illustrate the situation, and the annexe 9.4 presents a real example of an engine configuration.

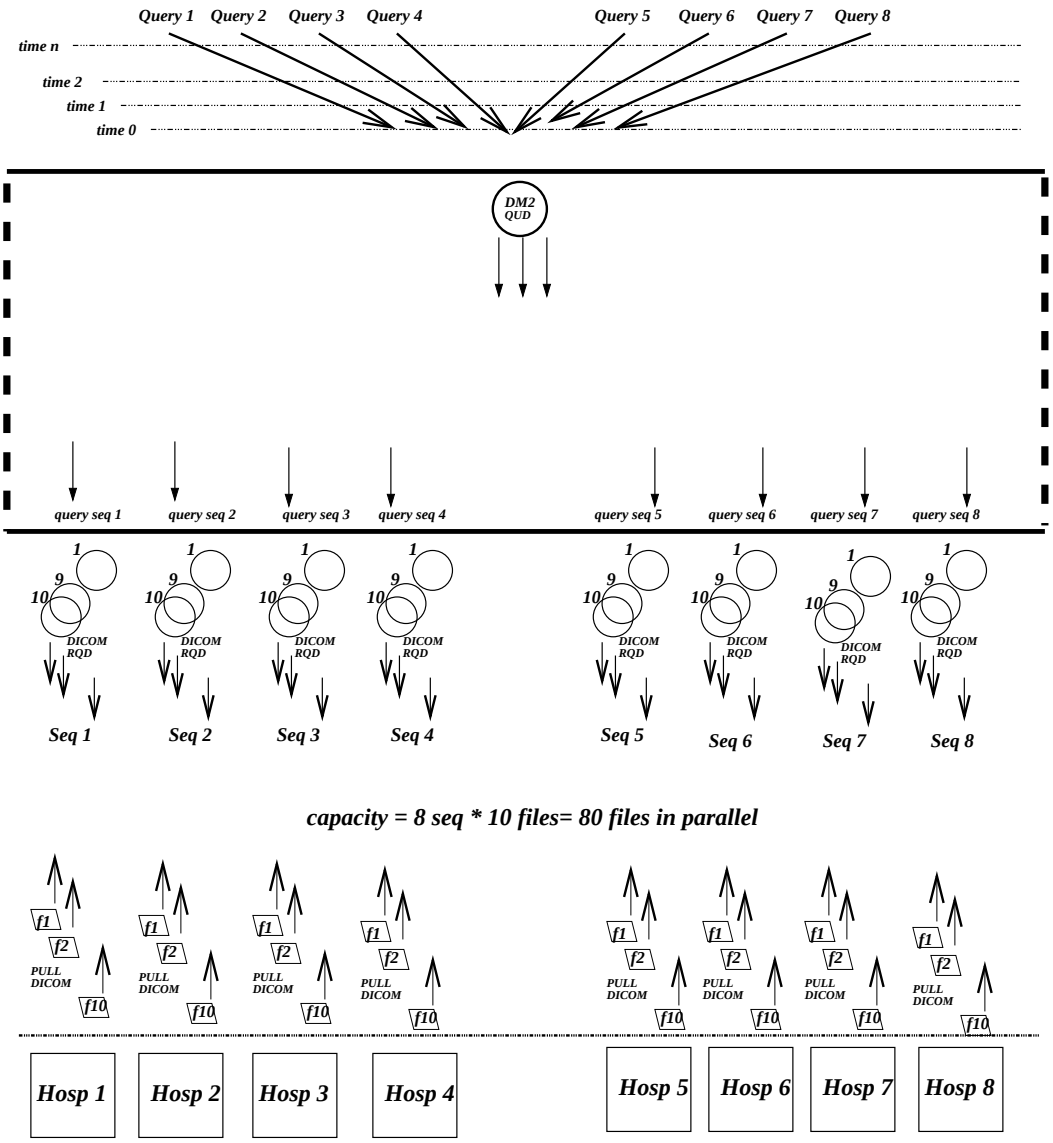


FIG. 6.1 – Saturation : Using all the capacity

The big square represents the server engine, and small squares are the hospitals (DICOM servers). The external arrows in the top side of the figure are the queries which arrive at the same time to the DM² QUery Driver (a circle). The external arrows in botton of the figure show the DICOM pulls of one multi-slice image by hospital. The internal arrows represent the queues of messages by each one of the ReQuest Drivers (RQD), which are designed as circles in the botton side of the engine. The time lines represent the moments in which the queries arrive to the engine ; thus, in this case, all the queries arrive at the same moment t0

We have stored test sequences of DICOM images into the 8 different simulated hospitals. Each sequence is made of exactly 10 slices (as shown in figure 6.12). This allows us to transfer in parallel all the slices of one image.

6.2 Performance Tests

6.2.1 Message Passing Test

As defined in the architecture chapter, the message passing layer is only used between local processes, so it is based on IPC mechanisms. A message passing tool such as PVM or MPI is based on remote communication and it is an additional weight for the DM^2 engine which can be unnecessarily costly.

To test these hypotheses, we compare the behavior of $DSEM^0$ to PVM. We made an experiment where a set of messages were transmitted back and forth (echo) between a client program and a server program which uses PVM in the first case, and $DSEM^0$ in the second case. We measured the response time (microseconds) of the round trip set of messages.

In the first (see figure 6.2) experiment we transmitted a variable number of messages (*500, 1000, 1500, ..., 5000 messages*), with a constant size of each message (15 Kbytes)[3]. In the second experiment (see figure 6.3) we delivered the same set of messages, but changed the size of the message (*15KBytes, 20KBytes, 25KBytes, ..., 110KBytes*).

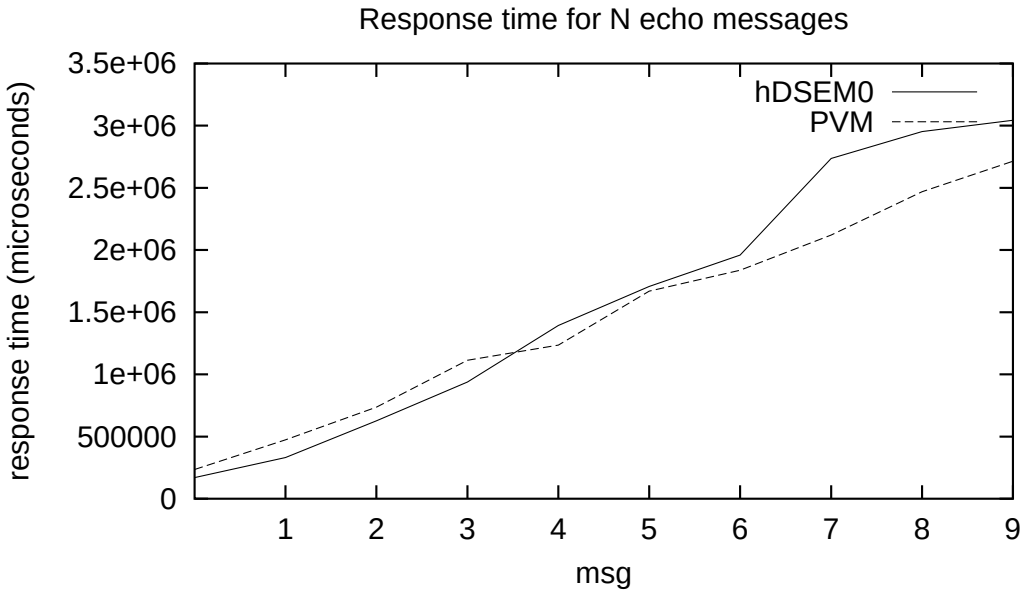


FIG. 6.2 – DSEM0 vs PVM
500, 1000, 1500, 2000, 2500, 3000, 4000, 4500, and 5000 messages of 15 Kbytes.

This experiment allows us to conclude that the behavior of $DSEM^0$ is quite similar to that of PVM (local communication) for a fixed message size of 15 KBytes (see figure 6.2). When the size of a message increases, the performance of $DSEM^0$ is better. Figure 6.3-(i, ii and iii) shows a 3D plot of the same data with different angles of view. The surface over, is the plot of PVM data, which means that response times are greater than the $DSEM^0$ ones (surface under). One can note that the distance between the two surfaces increases when the size of the messages also increases. The

[3]At the moment, DSE^0 manages an average message size of 15K.

best performance of DSEM^0 is due to the fact that it works exclusively on share memory, while PVM must deal with network operations. However, this is evident when the number of delivered messages increases.

6.2.2 Query for Retrieving an Image

In chapter 5 (section 5.3.1 - figure 5.8) we illustrated a query for retrieving an image (set of slices) from its DICOM storage. In summary, a request (*getDM2Image*) is made to a QUery Driver (QUD), and it : (i) localizes the image (in a hospital, another engine or cache), (ii) gets the image, (iii) assembles the image, and (iv) copies the image to the requester.

The query is resolved in six steps, as shown below. Each one of these steps is represented as one phase of the query, and then analyzed for its processing time. Thus, the query :

- looks for the image in the cache tool,
- uses the database (*metadata TKD*) service to locate the DICOM files from which the image must be assembled,
- looks for slices into the cache,
- copies the image from the DICOM server,
- assembles the image (set of slices) into a single file (by using the image tool),
- stores the image into the cache and return it to the requester (grid or user).

As described in section 6.1, the test system is able to transfer 8 images (each one of 10 slices) at the same time ; that defines the system capacity. A concurrent query of more images, will produce a queuing of queries.

First, we tested 16 queries [4] in parallel, to study the application profile. What we were looking for was the participation of each phase of the query in the final response time. These are hybrid queries as the ones described in section 5.3.1 and figure 5.8.

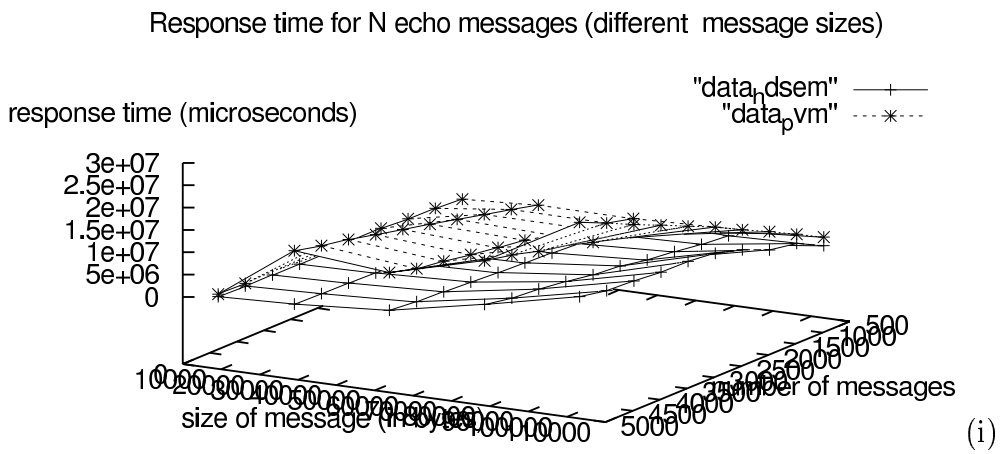
Figure 6.4 shows the percentage (Y axis) of the total response time by each phase of the query, for each one of the 16 queries (X axis). For example, in query number 2 (X axis), the DICOM phase has taken 50% of the total execution time.

In these processes there are services over which we have no control, such as those that follow :

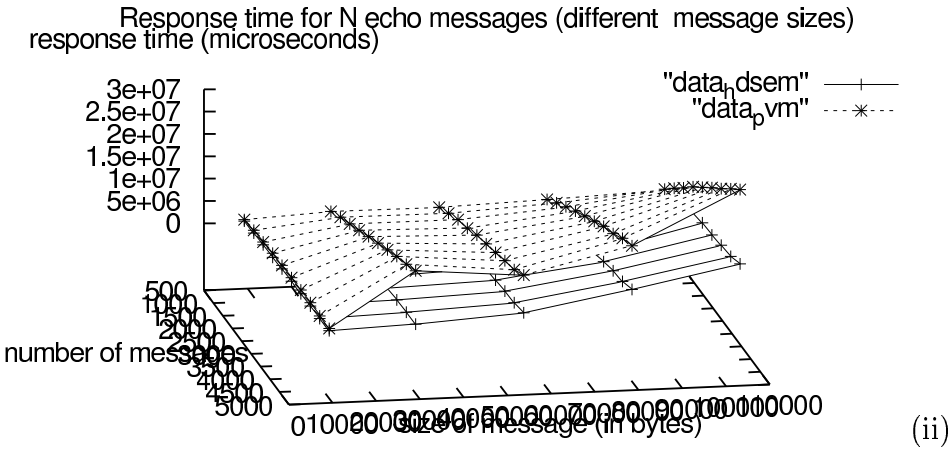
- Cache : we made this test with a prototype cache, but a superior one is being developed at LIRIS by our research team [187] [92] [93]. Its time consumption does not exceed 9%. Although the cache must be accessed up to 4 times, in this query we have plotted only the items *ii and iii* of the case scenario : (i) checks for answers to the hybrid query, (ii) checks for sequence of an image existence, (iii) checks for slices existence, and (iv) registers slices and sequences then into the cache.
- Security : we consider that the hybrid query starts once security was checked, so the response times do not include the time of security checks [5]

[4] When receiving 16 queries, the engine can process eight (the system capacity) of them concurrently and must queue the other ones for later processing.

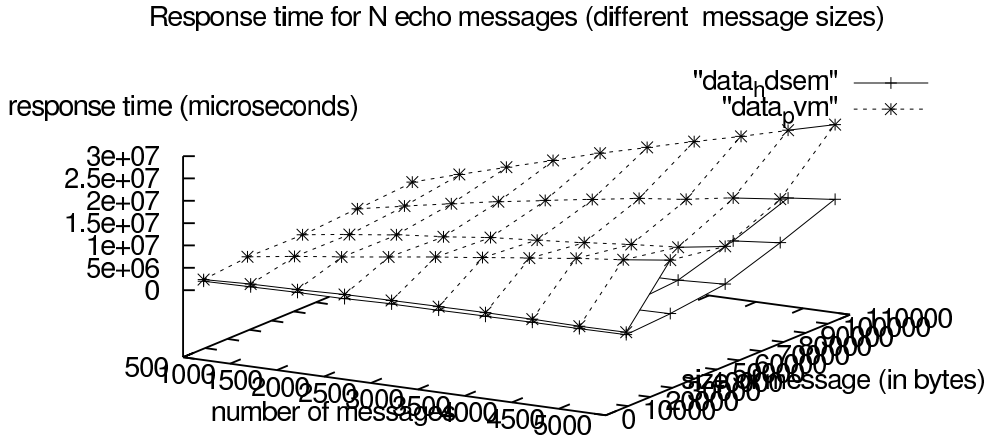
[5] Our research team [187] [90] [91] is also working on this topic.



(i)



(ii)



(iii)

DSEM0 vs PVM (not remote) for 500, 1000, 1500 ... 5000 messages of 25 ... 110 Kbytes. These plots are 3-D views generated by using gnuplot with coordinates as follow : (i) 60, 120, 1, (ii) 60, 80, 1, and (iii) 60, 30, 1

FIG. 6.3 – DSEM0 vs PVM.

– Grid : this time only concerns the registration of sequences of images, not the transmission of the images. Its participation is about 4%. Our meassure finishes before submitting a job to the Grid : the time of processing a submitted job is out of our control.

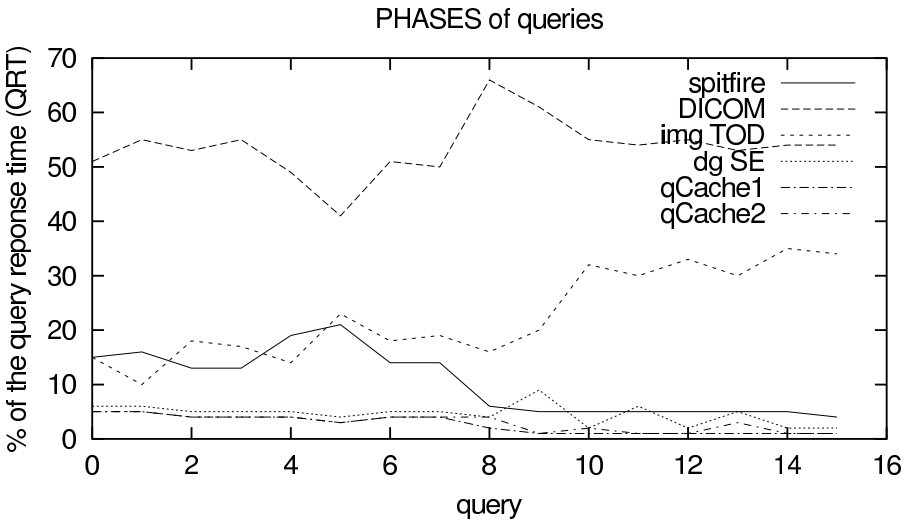


FIG. 6.4 – Hybrid Query : performance of each phase
Percentage of time for solving each phase into the hybrid queries number 0, 1, 2, ... 16)

The experiment shows that the main part of the time of the query is consumed by two phases :

– Images manipulation (about 25%) : It is a TOol Driver (TOD) which is in charge of assembling the set of DICOM slices into a single DICOM file. Performance could be improved by developing techniques of parallelism in this driver, but that scope is out of our research interests. Our architecture allows one to *change* this *TOD* by one which is faster.
– DICOM pull (about 60%) : This phase is managed by a ReQuest Driver (RQD) in charge of getting in touch with different DICOM servers (CTN) at hospitals. We cannot modify the DICOM server.

These execution time percentages are normal because they correspond with the time to access images and to manipulate them. The three following experiments address the DICOM phase of the query, because this access to data is the most time consuming phase.

6.2.3 Saturation Condition

We have designed an experiment to evaluate the behavior of the system when its available resources of data access are saturated. Figure 6.1 illustrates the testing conditions, and the annexe 9.4 describes the configuration details.

The experiment consists of *using the whole capacity of the system*, as described in section 6.1, so we issue concurrently 1 query (*getDM2Image*) for each one of the 8 hospitals. Because each query concerns a sequence of images with 10 slices, that

means that our concurrent queries stress the system to pull 80 DICOM slices in parallel, which is exactly the configured data access system capacity.

The figure 6.5 plots the total response time (Y axis) of each one of the 8 queries *from the initial time* ; however, is important to understand that the total time of the experiment is the one plotted for the last finishing query. In other terms, it shows the total time from the start of the experiment (*t0*), until the end of each query, *e.g.,* the first plot (6.5-i) shows that the experiment has a total duration of 16.5 seconds; the first query has finished 12.5 seconds after the experiment has started, the second 14 seconds after the experiment started, etc. This means that the whole experiment starts at *t0* and ends at *tn*.

The first figure (6.5-i) shows what happens when considering the access to disk. The second figure (6.5-ii) illustrate an experiment without disk unit, and the third (6.5-iii) is a comparison of the two previous curves.

We see in (6.5-i) a bottleneck in the access to the disk in the *server engine* [6]. The system has succeeded in pulling the 80 DICOM slices, but it tried to write (to the disk) almost everything at the same time, which produced a traffic jam. The plot shows the same response time from queries 3 to 8, but a different response time for the first two queries. What happens is that the system succeeds in finishing the first two queries [7] before the bottleneck in the disk, so all the other queries finish at the same time.

In figure (6.5-ii) the cost of writing to disk is not considered, so we can see a reasonable growth in the response time for these eight queries. It increases from 2.9 seconds to 4.5 seconds in the last query. The cost of concurrency is 1.6 seconds (50% of the first one). The system makes one query in 2.9 seconds [8] but the eight queries in 4.5 seconds.

The comparison of these two plots is shown in figure 6.5-iii. It is clear that the cost of the disk is important. The time for ending the first query, is four times more when writing to disk (2.9 seconds vs 12.5 seconds). It is similar for the total time of the experiment (4.5 seconds vs 16.5 seconds).

We consider that the system (*DSEM/DM*^{2}) performance is independent of the performance of the disks, once they can be changed for the highest technology without modifying the *DSEM/DM*^{2} system. However, this demonstrates clearly that when designing an operational system, attention must be put on the storage facility efficiency.

A more realistic situation, as the one described in section 6.2.5, does not present this problem. Queries arrive following a random access pattern, so the system has an advantage in resolving one query before the next one arrives.

Any way, these experiments clearly show : (i) that our system can cope with very

[6] All the sequences of images are transmited from the different hospitals to the server engine, where there is only a single shared disk.

[7] To finish a query the system must transfer up to 10 files in parallel, where each file transfer corresponds to an independent task ; so, two concurrent queries may have not finished even if many of their tasks (files transfers) have ended. For example, a query can complete 9 transfered files (of ten), but being waiting for the last one, which the operating system is writing to the disk yet.

[8] This response time was almost the same (2.75 seconds) for an experiment with only one query using the same machine and configuration. The difference to 2.9 seconds is represented in the advantage of having available the whole system in order to solve a single query without concurrency

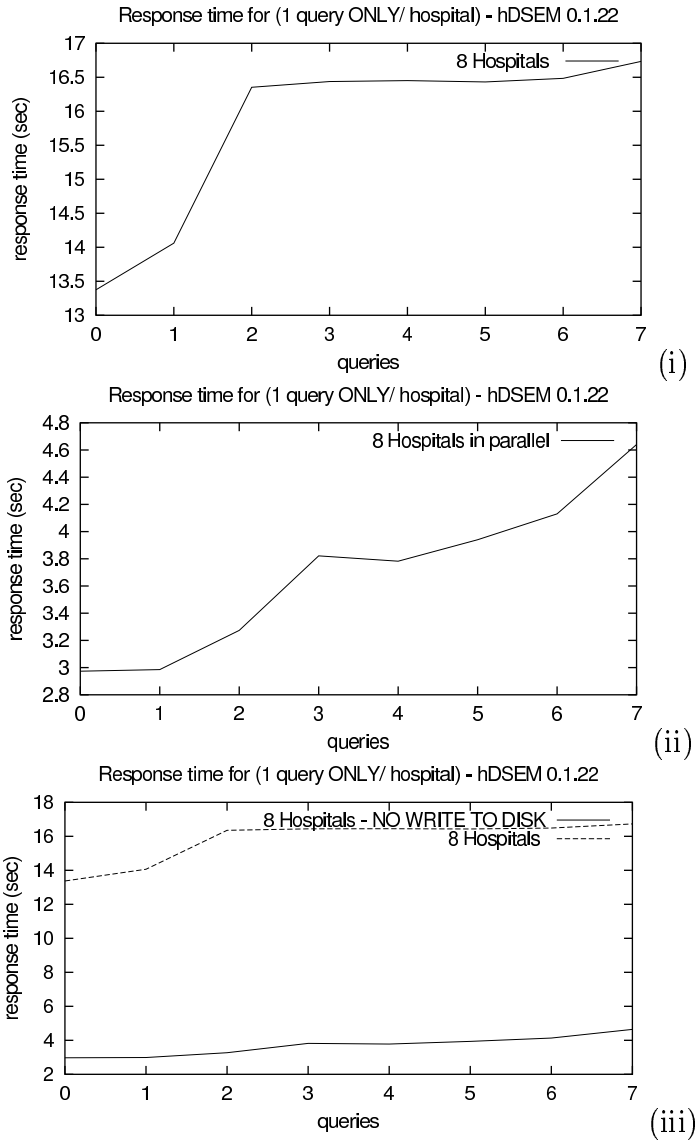


FIG. 6.5 – Saturation of the System (1)
8 hospitals in parallel ; one sequence by each hospital. (i) Considering write to disk. (ii) without write to disk. (iii) comparison.

streesing conditions, and (ii) that response times are compatible with physicians demands.

6.2.4 Overload Condition

The second experiment uses the same machine and data conditions as described above (section 6.2.3) but the goal is to *require from the system, more than its capacity for data access*. So, instead of one query by a hospital, we deliver 10 queries per hospital at the same time (concurrently). Figure 6.6 shows that these stressing conditions produce a queue in each one of the request drivers (RQD) for each hospi-

tal. The system is required to pull 800 slices [9] in parallel while the maximum delivery is 80.

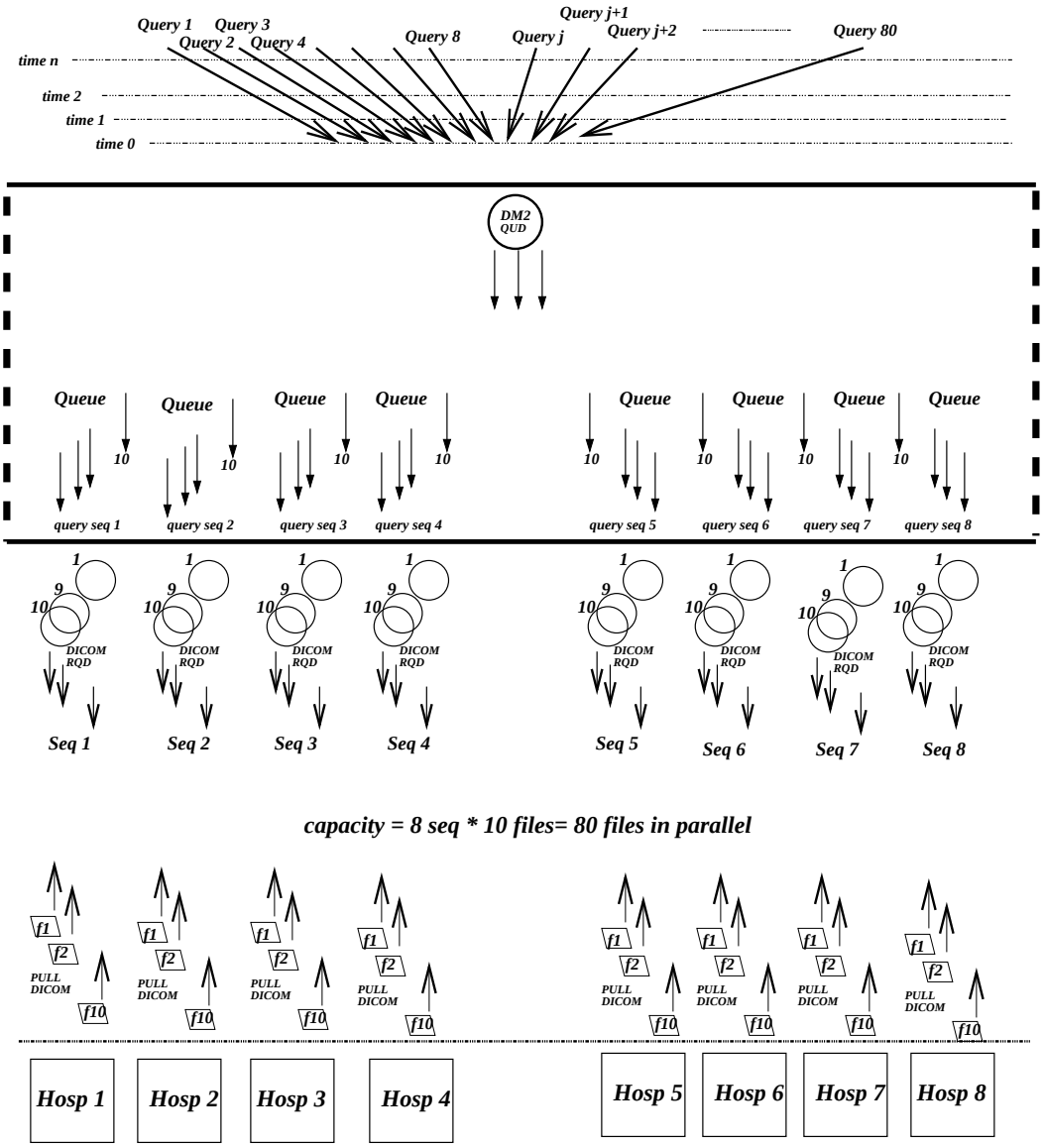


FIG. 6.6 – Overload
Using more than the capacity

Figures 6.7-i and 6.7-ii show what happens when the operation of writing to disk is considered or not . The behavior is similar to the one described in section 6.2.3.

6.2.5 Random Access Pattern

Sections 6.2.3 and 6.2.4 show conditions of saturation, but the reality is quite different : queries do not always arrive at the same time. We have run an experiment

[9] 10 sequences of images * 10 slices/sequence * 8 hospitals = 800 slices

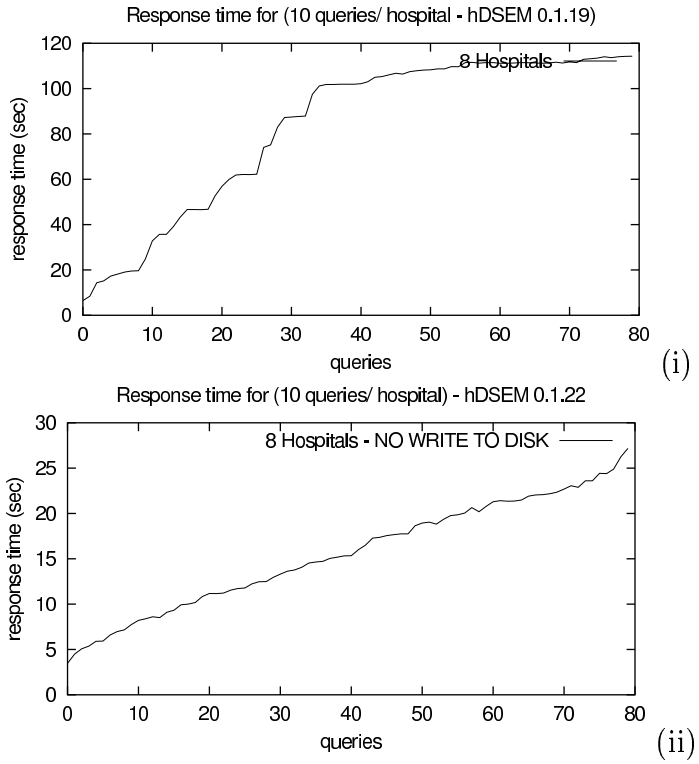


FIG. 6.7 – Overload
8 hospitals in parallel; ten sequences by each hospital

in which the queries arrive separated by a random delay whose duration follows a Normal or Poisson distribution of probability (poisson desviate [99] [101]) [10].

Figure 6.8 shows that : (i) the queries arrive to the engine at different instants of time (*t0, t1, ..., tn*), and (ii) the size of the internal queues by hospital (RQD) is different, depending on the arrival time.

We also measure the response time (considering access to disk) of each query, but we are interested in looking at its behavior, and at the queue size at different instants of time. In this experiment we aim at observing how the system manages its load without crashing.

In figure 6.9 we plot the response time for each one of the issued queries (10000 queries). In this figure, the time between queries was simulated from a random source with *Normal* distribution [11] :

```
mean     = 2 seconds
standard deviation = 1 second
```

[10] We use the Poisson distribution because it is classical in an evaluation of queue systems. The Normal distribution was used for comparison reasons : it plays a central role in statistics and has been found to be a very good model for many distributions that occur in real situations.

[11] From the experiments above, we found a response time of at least 4 seconds in the solution of a concurrent query ; so we selected a mean value of less than 4 seconds for guarantying that the system must solve more that 1 query at a time. The standard deviation of 1 second eventually allows an occurrence of two simultaneous delivered queries, which means, a time of zero seconds between two queries.

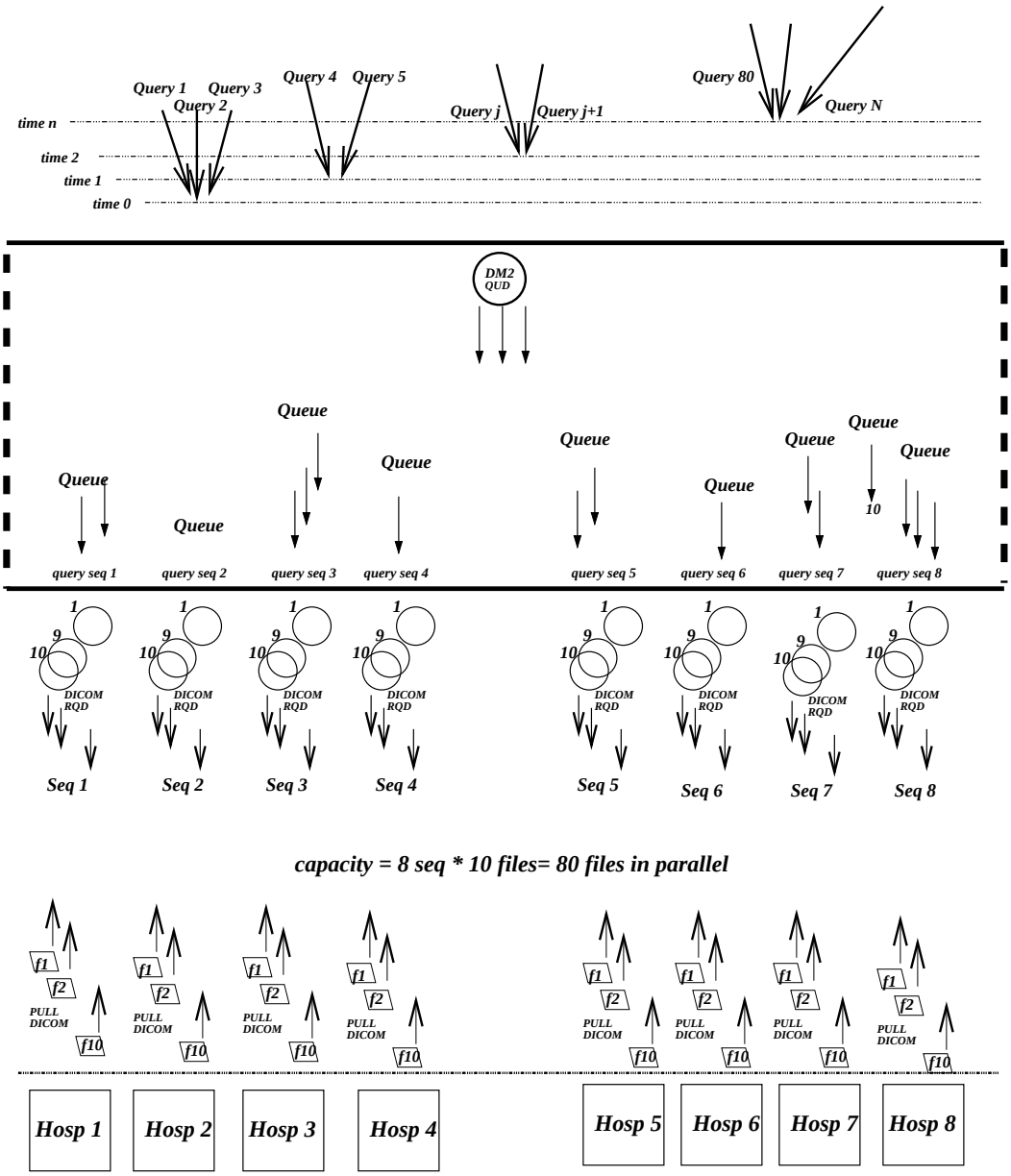


FIG. 6.8 – Random Source
random generation of queries

The figure 6.9 shows that in these experimental conditions the response time remains very stable and very acceptable (less than 7.1 seconds).

For the experiment illustrated in the figures 6.10-i and 6.10-ii, the time between queries was simulated from a random source with the distribution *Poisson* [12]. When using the distribution *Poisson* in our experiments we use a **mean of 2 seconds**, similarly as for the *normal* distribution. In figure 6.10-i we plot the size of the query

[12]We use the Poisson Desviate, which gives an integer value that is a random deviate drawn from a Poisson distribution of the mean. This is interpreted as a waiting time between events. The Poisson Distribution is ussually used for modeling rates of occurrence.

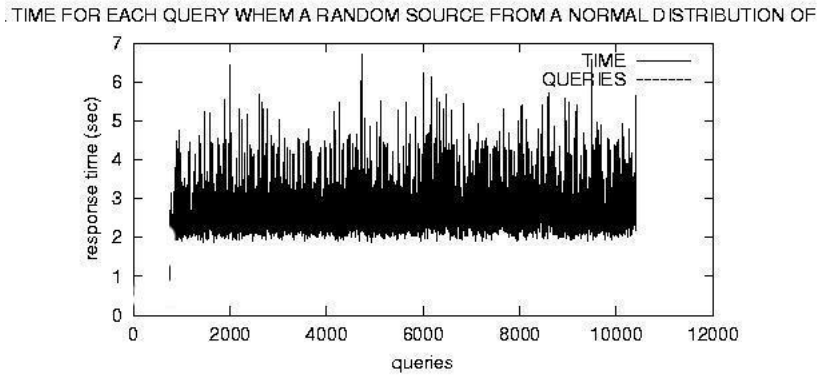


FIG. 6.9 – Stressing the System (Normal)
A source of random query events. The response time for each one of 10000 queries

queue every 3 seconds (1200 samples each hour) ; it plots the number of queries that the system has to finish (regardless of the hospital concerned) in an instant of time (ti).

We see that the system becomes loaded by instants (the queue or the response time increases), but the internal interaction between processes allows one to process them. In fact, we see a stabilization of the response time in the queue query size around an average value (2 queries).

In figure 6.10-ii we plot again the response time per each one of 2300 queries. Because queries do not arrive at the same time, the system has the possibility of dealing correctly (most of the time) with the access to the disk ; however, oppositely with a Normal distribution of the queries, here, with a Poisson distribution the problem of a bottleneck while accessing (described above) the disk can appear by short chunks of time. It can be shown in the impulses shown in figures 6.10-i and 6.10-ii. It is important to see that despite these bottlenecks, the system can recover itself and continues running normally.

In fact, figure 6.9-i shows that the system rarely has more than 4 concurrent queries waiting to be solved (queue), which is quite different than the cases in sections 6.2.3 and 6.2.4 ; additionally, those queries are being processed in different phases, because they arrived at different moments. This last sentence means that, even if there are 4 or more concurrent queries, one could be in the DICOM phase, while the other is being processed in the database phase.

In conclusion, these first series of experiments show the efficiency, the scalability and the robustness of the DSEM/DM^2 system. Under very stressing conditions, the system never crashed and gave very effective response times. Finally, these experiments demonstrate the feasibility of the system both in terms of its capability to cope with high numbers of queries and its compliance with physicians constraints in terms of response time.

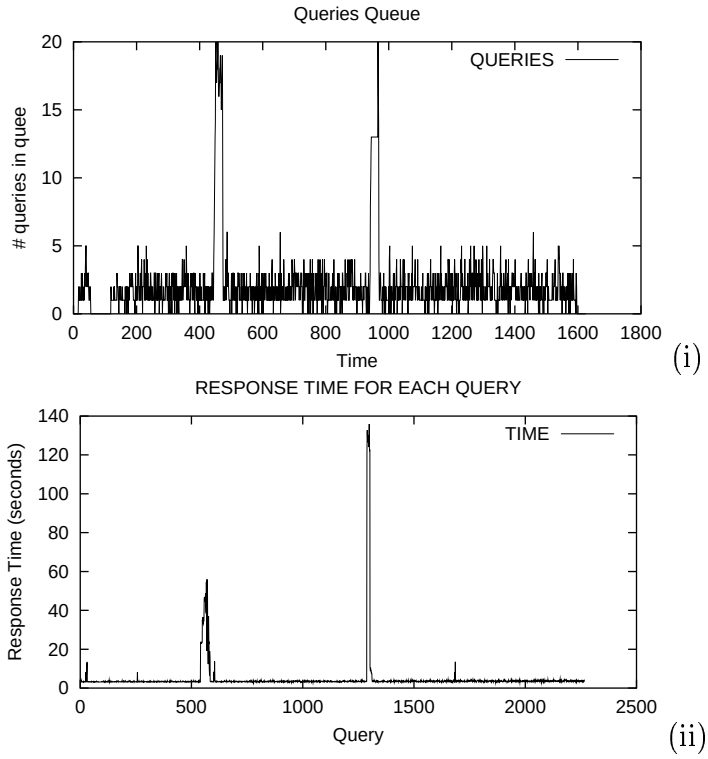


FIG. 6.10 – Stressing the System (Poisson)
A source of random query events. (i) size of the queries queue every 3 seconds (1200 samples each hour), (ii) response time for each one of 2300 queries.

6.3 A Medical Distributed System

So far we have proposed an architecture *(DSE)* for building distributed systems which address the problems of the Medical Community, and have developed a prototype from that architecture. Our implementation has separated the middleware components *(DSE^0, DSE^1, DSE^2)* from the application components *(DM^2)*.

In chapter 5 we addressed the implementation issues and illustrated the DM^2 from the server point of view : we presented the main queries and showed how they are solved by interacting with different components (see section 5.3.1). Chapter 5 presented a higher level view of the DM^2 system, by placing emphasis on the services. In this section we present a medical application which uses the DM^2 system, and which we have tested as a prototype implementation. This application is presented from the user point of view.

In our implementation of DM^2 we offer the API3 for communicating with a DM^2 engine. This API is useful in building interfaces (*e.g.,* GUI) able to offer access to the DM^2 services. We have not yet implemented graphical interfaces, but we have implemented a command line interface [13] which allows testing and using all API3 functions. Therefore, we present below some of these commands in order to illustrate the use of DM^2 (sections 6.3.3, 6.3.4 and 6.3.5) ; however, what we want to illustrate

[13]The command lines are integrated with the API3, which is available on the Internet. The link is in 9.7

are not the commands by themselves, but we aim at present the different information which an user can obtain from the DM^2 system.

In other words, chapter 5 put the emphasis on the 3rd layer (application), and here we address the 4th layer (user). Additionally, we summarize the whole DM^2 system in section 6.3.1. The examples illustrated in this section were all ran by using the DM^2 prototype, except the ejection fraction algorithm which is still being developed, but once finished can be executed by DM^2 as any other one algorithm. The environment details as well as the configuration ones, are presented in the annexes chapter (9).

6.3.1 Overview of the DM^2 system

A DM^2 engine is the brick for building a medical system which provides secure and confidential access to the medical data sources. It provides access to medical data stores and to external services such as Grid resources, and also offers services of high throughput computing by using a Computing Grid. These possibilities should facilitate research on pathologies and epidemiology, allow researchers to assemble *virtual data sets* suited to statistics extraction or study of rare diseases, increase the scale of experiments to levels not reached before, provide the access to image processing services, and ease the access to such image processing tools for the end user.

However, such a system is far from being completed today. Although there has been enormous progress, Grid projects are not very stable environments yet. Security and confidentiality for medical data are subjects of research, and the access regulation is still a barrier. Encouraging the medical community to share its data is another issue that is facing a human barrier. Despite these challenges, our *engines* address some of the problems to be solved (see section 4.8.2) and allow us to implement a prototype system, as described in figure 6.11, and to test some parts of it.

The whole system is composed of a set of DM^2 engines which are divided into *DM^2 Server Engines* and *DM^2 Client Engines*. The Server Engines are in charge of a geographical region (Lyon, Toulouse, Bordeaux, etc) and have communication with the other regions (Server Engines also), and also with a Grid in order to have access to computing and storage resources. Each *Server Engine* hosts different *hospitals* and *end users* (physicians, health centers, researchers, etc). Hospitals are implemented with different pieces of software (CTN [135], DCMTK [136]) for dealing with medical data , with a *DM^2 Client Engine* which ensures connection with the region server, and with medical image devices (Magnetic Resonance Imagers -MRI-, 3D Computed Tomography scanners -CTscan-, PET, X-RAYS) all DICOM3 compliant. Therefore, each region has a hierarchical structure from the scanners up to the *engine server*, through the *client server*.

The raw data are acquired by the scanners, but they become structured (with strong semantics and metadata) once imported into the system. End users get connection to the system and make hybrid queries over this data, which means, that they submit queries that need not only access to the metadata, but also to the image's content. This second part of the query, also requires computing in order to

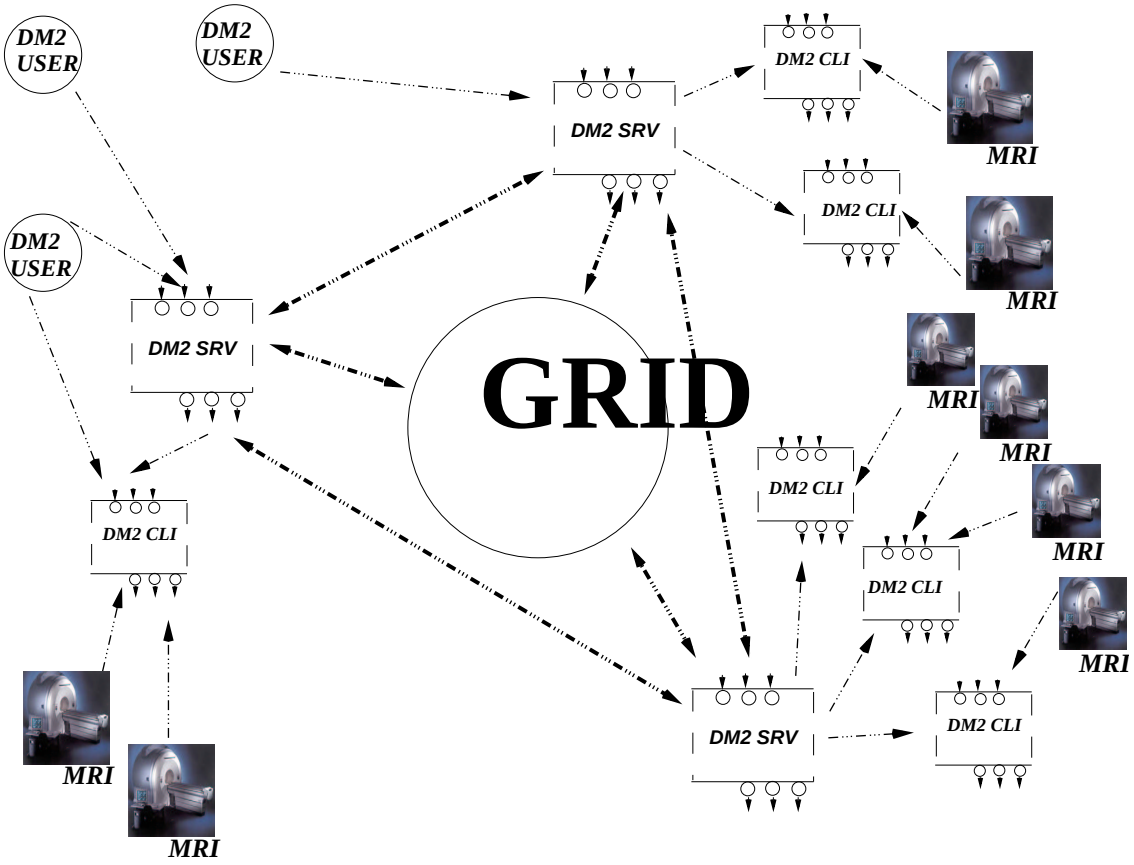


FIG. 6.11 – A DM^2 System represented as a set of DM^2 Server Engines and DM^2 Client Engines. MRI devices are the raw data acquisition point.

process the query. A user can add new data to the system, such as new processed images or relationships between images and results.

Configuration examples for a *DM^2 Server Engine* and a *DM^2 Client Engine* are shown in the annexes 9.1 and 9.2. Sections 6.3.3 and 6.3.4 show examples of medical applications. The section 6.3.2 shows the data acquisition mechanism.

6.3.2 Image Capture

We have installed a *DM^2 Client Engine* at the Cardiological Hospital of Lyon in order to manage the image data capture process. The internal design of this *engine* was described in chapter 5. It deals with : (i) the registration into the DM^2 Server Engine and into the Grid System, of each slice in the sequence of images, (ii) the extraction of useful metadata and their transmission to the DM^2 Server, (iii) issue of a query to the Server in order to process the images which are being captured ; the processed information can be used as a higher class of metadata (*e.g.*, histograms, texture information, etc [14]).

[14]The DM^2 includes image processing algorithms. Those algorithms are matter of research at the CREATIS laboratory and DM^2 provides the mechanisms for applying these algorithms to an archive of images. We have tested some of them, as is shown in this chapter.

Image data are acquired from Magnetic Resonance Image (MRI) devices and stored in DICOM3 format [49]. The MRI use the DICOM-net part of the DICOM standard in order to transfer (DICOM push) the sequence of images to a DICOM server which extracts the in-file metadata and stores it in databases where they can be later exploited. For this purpose, we have installed CTN [135] at the Hospital [15].

One problem to deal with, is the N-dimensional nature of the images, and especially, their complete registration into the DM^2 system. As described in chapter 2, a 3D cardiac sequence of images represents a 3D volume evolving over time. The image 6.12 shows seven sections of a heart along 30 instants of time. Each one of those sections represents a 3D cardiac volume, but is composed of seven slices (2D images). So, this sequence of images has 210 slices [16] which *must be independently registered into the DM^2 system but also must exist as a unique image having a relationship between them.* Those 210 images are called slices, and each one is stored as an independent DICOM file. One of the things that DM^2 does is to deal with that set of files but also builds an assembled version of the sequence in *only one file* ; in this way, the storage details become transparent for the end user and the Grid.

One difficulty to be resolved was how to automatically intercept the DICOM push which is done from the MRI, and to produce an action into the DM^2 system. A DICOM push is used to transmit images to a DICOM Server (*e.g.*, CTN or a PACS) from the scanner console : the images are not pushed to DM^2. So, producing an automatic action into DM^2 means modifying external software (CTN, or the PACS, or the MRI console software). We were looking for a more general solution because we did not want to modify other systems.

The second concern, was how to know when a sequence of images was completely transmitted; which means, how to find out that the last DICOM push (*e.g.,* the 210th slice or file) corresponds with the end of the sequence of images. The third problem was to know the exact order of the images in the sequence.

The solution was provided by DCMTK, because it has mechanisms to automatically execute line commands of each new image, and to execute additional commands when an end-of-study (DICOM field) or a timeout (seconds) is determined. So, in addition to CTN, we have installed the DCMTK [136] DICOM Server as a wrapper between CTN and the DM^2 Client Engine. As described above, this wrapper is a way of dealing with the registration of sequences of images and also a way of automatically capturing the scanning events (DICOM push from the MRI). The MRI scanner transfers images to DCMTK instead of to CTN, and then, DCMTK forwards those images to CTN. What DCMTK does additionally, is to execute DM^2 commands in order to import data and metadata into the DM^2 system (see the figure 6.13 and annexe 9.3 for details of configuration).

DCMTK can deliver DM^2 commands each time that a new file is registered into the DICOM server. Indeed, it is possible to apply an algorithm list to the captured images. For example, in the database we can define the list of algorithms to be applied for each one of the captured images of the heart by a particular radiologist.

[15] The Cardiological Hospital of Lyon is considering to install PACS and RIS systems. The PACS archive the images and allow image transfers. The RIS contain full medical records : image-related metadata and additional information on the patient history, pathology follow-up, etc.

[16] 7 2D images by section x 30 instants of time = 210 slices or files

For example, we can define the algorithms *mean* and *histogram* to be applied to a set of images; thus, these images are pre-processed in order to get their mean and histogram. These are new high level metadata which are kept in the database and can be used in order to reduce the search domain for future queries. A Computing Grid helps by providing the required resources in order to pre-process this sequence of images, when necessary [17].

6.3.3 Similarity

Physicians are often interested in looking for medical cases similar to the ones they are studying (see section 2.3 in chapter 2). A case may be identified as "similar" if the image content is similar (*e.g.,* the same region of the body, the same kind of acquisition) but this is often not sufficient to discriminate between a set of acquisitions. An alternative way to compare medical images is to use similarity measures [28]; for instance from the gray level intensities in the images, one can compute several measures [44] : (i) simple differences, (ii) coefficient of correlation, (iii) Wood's criterion, (iv) correlation ratio, (v) mutual information or entropy, etc - see annexe 10.1.

Given an image of interest (source image), a hybrid query can be executed as a pre-selection, first, of possible images to compare, and then an image content-based analysis on these images is run to search for most similar images. The pre-selection is based on SQL database queries, and aims at decreasing the domain of comparison; metadata such as the image region, modality, or orientation can be considered, but also pre-computed high level metadata (if available), such as histograms of intensities. Figure 6.14 shows a desirable result for the process.

Although each measurement is not necessarily very compute intensive, the comparison of a sample image against a complete database is intractable, in a reasonable time, on a single computer due to the size of medical databases. The actual cost of such a computation depends on several parameters such as the input image size and the computation precision desired.

Similarity Usecase

Dr. DELON looks for heart images similar to the one he has just acceded in order to confirm his diagnosis. He wants to rank existing images through a similarity score resulting from a computation involving his patient image and an image database. Once the images are ranked, he needs to visualize the most similar cases and their attached diagnoses.

First of all he queries the DM^2 System in order to get a list of heart images (*region="HEART"*) of his patients *(radiologist="P. DELON")*, which where acquired by a M.R.I device *(Modality="MR")* [18]. This is information in the DM^2 client database and is accessible by executing a line command (which we did not consider

[17] Only if the algorithm to apply is time-consuming

[18] A good criterion of pre-selection could be, for example, the *ejection fraction*, which would be a pre-computed metadata, but is an algorithm been developed (at CREATIS), yet. So, we could select the sequences with an ejection fraction greater than 0.5 (*e.g.*)

worthwhile to describe here) or by executing (through a GUI) a function of the DM^2 interface : both of which use the API3 in their internals.

He receives a response with a list of DM^2 sequence of images, which he requires to compare with another image he just registered in the system. For this purpose Doctor *P. DELON* chooses the similarity algorithm' of his preference (*e.g. "Montagnat" implementation [44], using the "wood's criterion"*).

Let us suppose that we have 8 sequences of images, and that each sequence has 210 slices as show in figure 6.12. The DM^2 Server Engine receives the query and retrieves the sequences (which means 1680 slices [19]) from the hospitals or from the cache (to improve latency). The *engine* translates the DM^2 file names into Grid file names, and then schedules one job for each comparison, into the *Computing Grid*. The Grid returns the results to the DM^2 System, which are backwarded to the *End User - (Dr. DELON)*, who receives a measurement of similarity for each one of the sequences of images..

In the case of the cardiologist who would prefer to validate the results, he can re-submits the query but using another algorithms' implementation (*e.g., "Pauna-Clarysse" [94], choosing the same "wood's criterion"*).

When he asks the system to visualize the sequences, slices are assembled in a single 3D image that is returned to the cardiologist for visualization.

The diagnosis he makes for his patient can be stored in the information system enriching the global knowledge, as well as the patient record. The diagnosis becomes available in a text file, so it can be stored in the database.

6.3.4 Segmentation of Cardiac Volumes

The knowledge of the heart anatomy and its functions is fundamental for studying ischemic diseases. Recent research [95] at CREATIS has addresses the problem of extracting the heart anatomy from the magnetic resonance cardiac images by using deformable elastic template techniques. The algorithms are based on a deformable model composed of topological and geometrical characteristics of the two ventricles of the heart ; however they are very processor consuming.

The *det3* [95] algorithm aims at segmenting (3-D) the right and left ventricles of the heart, from multi-phase, multi-slice M.R. Images. This means automatically extracting the pericardium and endocardic surfaces, in order to allow further clinical measurement of parameters.

This will allow one to generate a 3-D model of the heart having its morphology and functionality. The knowledge of the cardiac muscle bio-mechanics allows one to better understand the physiological reality and is very useful in clinical practice to evaluate the functional state of the ischemic heart. The figure 6.15 shows a Segmentation of Cardiac Volumes.

Segmentation Usecase

Dr AUTEUIL is looking for sharing with the scientific community the data which he has collected about morphology and the function of the heart. In order to do that,

[19] 210 slices x 8 sequences = 1680 file transfer in parallel

he is building an anonymous database with 3-D DICOM sequence of images (multi-phase, multi-slice) of the two ventricles, acquired in a M.R.I.

The following sequence of operations has to be performed : (i) interpolate the sequence of images into a single 3-D volume file, (ii) calculate the external forces which derive from the image, to be used into the deformable model, (iii) produce a 3-D segmentation, (iv) compute a measurement of ejection fraction, and (v) keep the output (text files containing the geometrical model) and make relationships with the source image. The former two steps are computed by an algorithm called **interpolation** [95], the third one by the algorithm **det3** [95], and the forth by the algorithm **ejectionFraction** [95] which is still being developed at CREATIS. The ejection fraction, as well as the produced output files, will become high level metadata, will be be stored into the database, and used later in order to reduce the search domain over the image database.

The algorithms are very time consuming, thus a high through-output computing resources are required. Dr. AUTEUIL makes a DICOM push of the sequence of images, from the console of the M.R.I. device (as it was described in section 6.3.2), so the information becomes available to the DM^2 System.

The algorithms are registered into the DM^2 System. Dr AUTEUIL has registered an *Algorithm List* number to be applied to all the images he has acquired. Thus, DM^2 applies the algorithms in the registered list to each one of the captured sequences.

Eventually, he receives additional images from his colleagues for inclusion into the database. Dr. AUTEUIL has an alternative process for including the images into the system, and producing high level metadata. He can register the sequence of images into the DM^2 system, and then apply different algorithms (an algorithm list) to that sequence. As an illustration, the commands below can be automatically generated by a user interface in order to allow Dr. AUTEUIL to make the interpolation (*interpolation algorithm*) of the DICOM sequence [20] before applying the segmentation (*det3*) and the ejection fraction computation [21].

```
./dm2cmd -inqalgo interpolation
-ids
1.3.46.670589.5.2.14.2198403904.1062079142.135100
-arguments
slice001,
slice210,
1.3.46.670589.5.2.14.2198403904.1062079142.135100.vol

./dm2cmd -inqalgo det3 -ids t0.par -arguments 0

./dm2cmd -inqalgo ejectionFraction
-ids
1.3.46.670589.5.2.14.2198403904.1062079142.135100.vol
-arguments
0
```

[20] Remember that this sequence of images has about 210 slices

[21] The file *t0.par* contains all the required parameters for running the *segmentation algorithm*, including the image file names. The first argument, set to 0, means no visualization (because a non interactive algorithm is waited for DM^2).

The segmentation process produces its output as text files, which are directly related to the processed image. Thus, the Dr. AUTEUIL can keep the results and his diagnosis in the database.

6.3.5 Second Similarity Usecase

The Dr. DELON has been realizing that it would be better if he only considers, for similarity comparison, the images in the database which are in a range of 10% of the source image ejection fraction [22]. This means that if his patient's image has an *ejection fraction of 0.5*, he will compute similarity measures only for the images into the database which have a ejection fraction of *05+-0.05*.

In this way, the ejection fraction (pre-computed) is used in a basic database operation (*SQL select*), and considerably reduces the operations to be done for the query.

What Doctor DELON has to do is : (i) to compute the ejection fraction for its patient's image, (ii) to query the database for getting the images with an ejection fraction within the specified range, (iii) to apply the similarity algorithm to the image list that he gets.

[22] Being developed yet

–

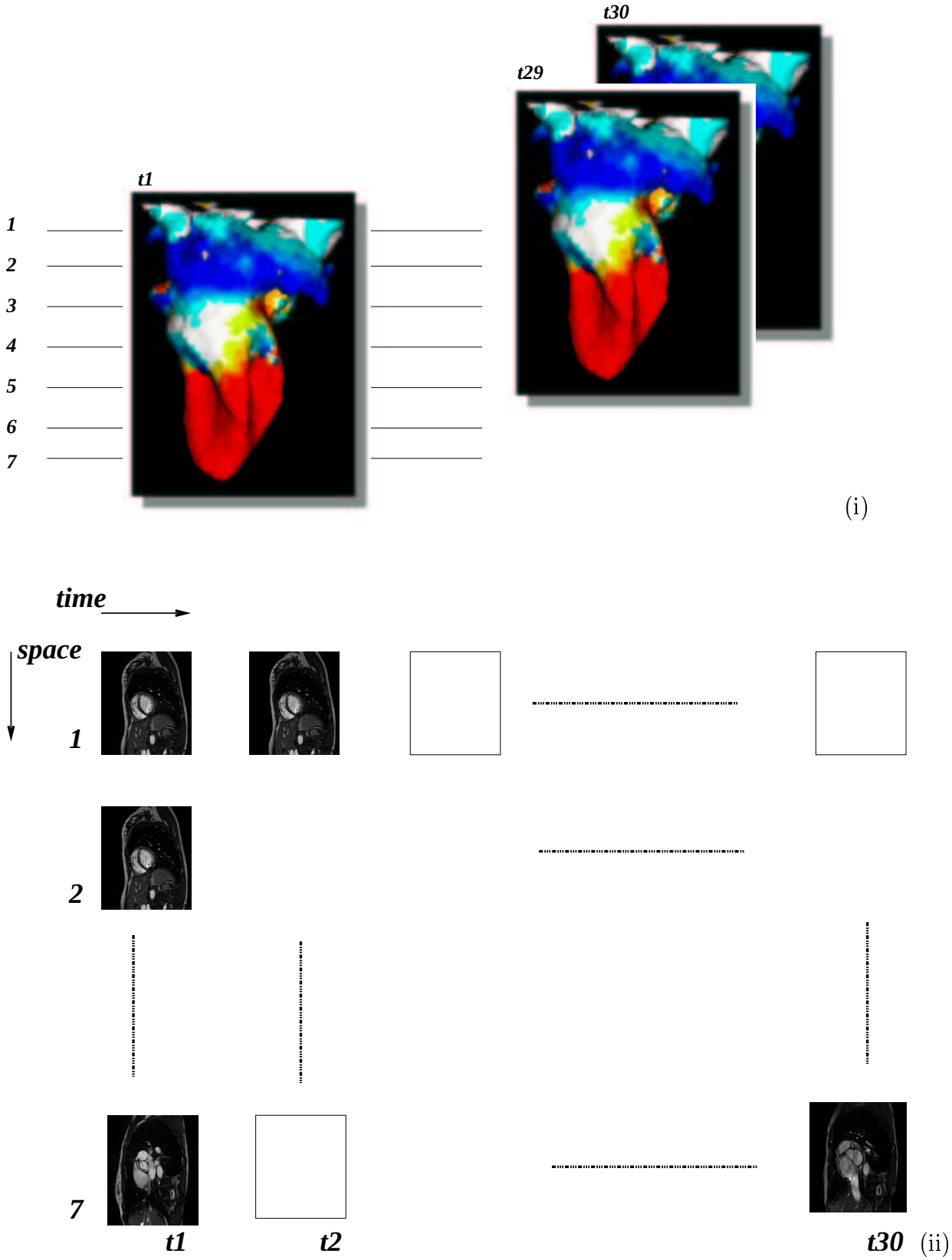


FIG. 6.12 – Sequence of DICOM Images

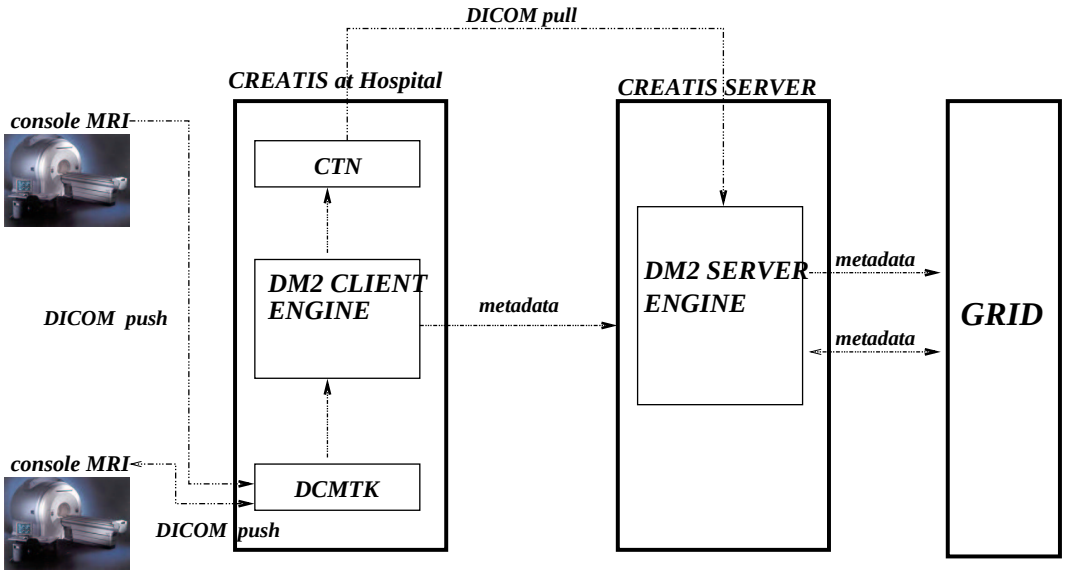


FIG. 6.13 – Hospital and Server

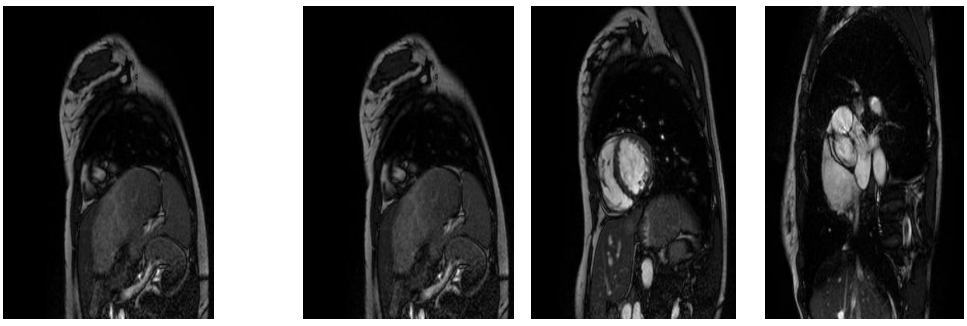
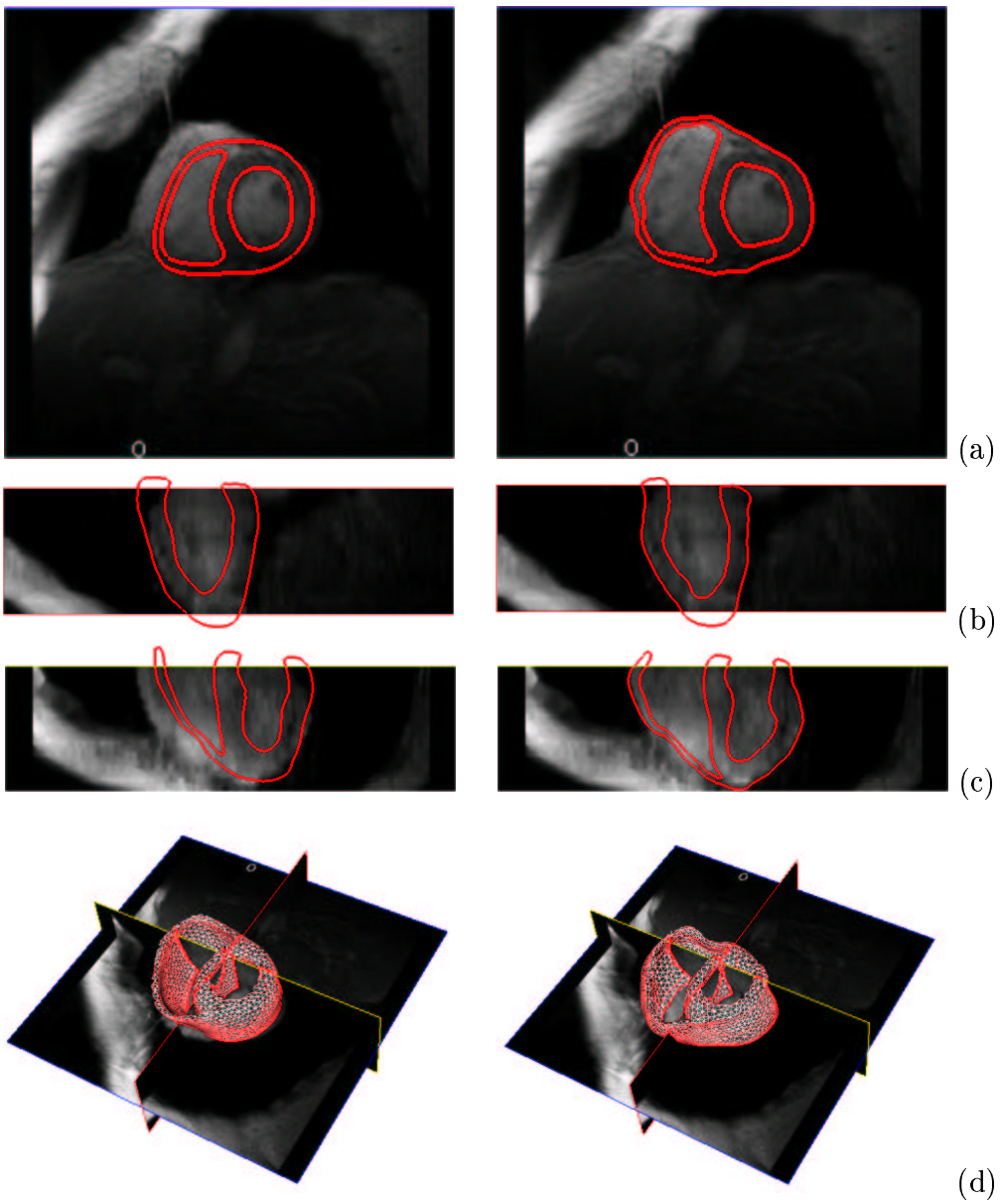


FIG. 6.14 – Similarity from left to right : source image and matching images with the highest to the lowest score. The image on the left side is the most similar to the others/



Left column corresponds to the initial model, and the right column to the deformed model. (a) orthogonal slices to Z, (b) and (c) orthogonal slices to X and Y, (d) 3-D images.

FIG. 6.15 – Example of a segmentation result using the 3-D deformable elastic template.

Chapitre 7

Discussion and Perspectives

"It's going to take a little while for businesses to be confident that grid computing works for them.", ***Irving Wladawsky-Berger****, IBM, USA*

–

In this work we have proposed a high level architecture (*DSE - Distributed System Engines*) for building *Distributed Systems (DS)* with the constraints of high-performance, high-throughput, and data-intensive management. This architecture has provided the basis for the design and implementation of a prototype of distributed medical image query system(DSEM/DM2).

Although our main target application is the querying of large distributed medical image datasets by their content, the DSE framework is applicable to any DS showing these constraints.

Our vision is that of a coupling between :

- a grid middleware used both as resource provider (computing power, storage, sensors...) and integration framework offering tools for interoperability, security and privacy enforcement, information sharing ;
- distributed entities (the so-called *engines*) in charge of interfacing the grid platform with decentralized (medical) information systems (e.g. medical image databases or patient record stores).

The DSE architecture has been specially designed in order to offer flexibility, extensibility and performance. These features are directly related to the specificities of health networks and applications dealing with medical information. Medical information is semantically complex, there is no universally accepted standard (DICOM excepted), correlated data (e.g. the patient record) are disseminated over multiple administrative entities with very different constraints, practices and policies (e.g. University hospitals, family doctors, small care units, regional hospitals, etc.), the number of actors is very large, end-user requests are of various types (epidemiology studies, medical diagnosis, education...).

In sections below, we summarize our main proposals and contributions, discuss their pertinence with respect to medical applications and to the implementation of medical grids, and present perspectives.

7.1 The DSE architecture

The *Distributed System Engines (DSE)* architecture eases the design of Distributed Systems at a higher level. Its generic multi-layered structure enables adaptability and scalability by differentiating the middleware issues from the application ones. In the same way, the horizontal definition of each layer in different types of drivers enables extensibility, openness and robustness.

Existing architectures such as MDA/OMA [191] propose a general vision of a Distributed System (DS), but do not define a base generic component : we do not only propose a model of DS, but we also define the logical structure a *functional brick*, the *Engine*, on which we propose to build DSs. Oppositely, higher level frameworks like MDA/OMA focus on the global interface and integration layer.

Thus, service integration standards as **CORBA** or **DCOM** define interoperability between distributed objects, but do not directly address such problems as high performance, access to sensitive data, massive storage and high throughput. They are more focused on the interoperability than on the architectural definition of a Distributed System, specially a large distributed system targeting complex information

sharing and high performance.

We defend in this thesis that interoperability mechanisms should be considered at an operational level rather than at the DS design level.

So, if desired or needed, the implementation of a DSE can have some elements compliant with interoperability specifications or middlewares (CORBA, DCOM, Java/RMI, OGSA, WSRF, DICOM...). This is a work to be done depending on the specific application environment on which the engine will be deployed.

7.2 Integration of Resources

Computing and Storage

A *DM²* *Engine* is seen as a set of cooperating application services, not as a resources provider ; in fact, an engine implements its services by using (transparently to the users) external resources like high computing power and massive storage. For doing this, it can use a *Grid* as a partner.

Distributed Engines (*DSEs*) used in the implementation of a Medical Application (DM²) like the one described in section 5.3 have a two way communication interface with the Grid. They can interface the Grid in order to get access to its resources (computing and storage), or can be interfaced by the Grid in order to give it a data-intensive access to (large amounts of) medical data registered in the Grid but actually stored within the hospital.

We have tested our prototype by interfacing it with *MicroGrid*, and in a first phase of the project with the *DataGrid (DG)* middleware. We implemented ReQuest Drivers for getting access to the *DataGrid Database Service (Spitfire)* and defined also an API with a *DG Storage Element*, but that software was not enough stable to allow us go on in the integration of these middlewares. Anyway, these various experiments demonstrated the facility of adapting an engine to any Grid environment.

Actually, many Grid projects (e.g. EGEE) use Globus as their core middleware. In that perspective, we plan to make DSE OGSA- [137] or WSRF- [163] compliant. In this way, *DM²* will become a grid service interoperable with other services and will be able to take advantage of all resources and functionalities available on the grid.

The framework proposed in this thesis has been designed to develop systems based on the *integration of services and tools* within a distributed environment. We have more specifically focused on the *access to data, registration of data and metadata management* in the context of *Content-Based and Hybrid Queries of Medical Images*. The issue with this type of queries is twofold : (i) queries require an important computing power in order to be executed, and (ii) (sensitive) data to be processed are distributed at a large scale. For instance, a single *similarity query* issued by a user can produce thousands of image comparison sub-queries.

The interconnection of *DM²* and the Grid offers an very powerful way to register, localize, retrieve, transfer and efficiently process such large amounts of distributed medical images. Without the computing power and the large scale integration facilities provided by the grid, implementing such an application would be untractable.

Furthermore, *DM*^{2} engines can provide storage facilities to end-users/applications by taking advantage of Grid storage resources, e.g. *Hierarchical Storage Managers (HSM)* such as Castor or HPSS. For medical applications that manipulate huge volumes of medical image data-sets, this facility offers a unique way to get access to very expensive storage technologies unaffordable for most of the medical institutions.

Access to data

The structure of Drivers, and specially the *ReQuest Drivers (RQD)*, allows an engine to be simultaneously connected to multiple instances of various database managers. For instance, an engine can access (low level) different hospital databases even if they are implemented with different database managers (MYSQL, ORACLE, etc). At the same time, it can also access a Grid Database Service such as Spitfire [1]. This makes possible the implementation of management tools of heterogeneous and distributed medical metadata. This feature also makes possible the "factorization" of multiple health actors under a single DSE manager. This is an important characteristics in a context where not all the actors have the possibility to host a grid access point (for security restrictions or a lack of technical competences or other reasons).

In the framework of biomedical grids, *DM*^{2} allows one to access to and to manipulate distributed medical image data sets and metadata bases. It also offers these resources as a service which can be invoked by the users of the Grid. In this way, DM^2 engines provide end-users with an access to external medical data and metadata which for security reasons can not be today permanently stored in the Grid data space.

DM^2 can be built on top of a *Distributed File Systems* such as AFS/DFS. Distributed File Systems allows one to store and access to data at a very large scale (up to thousands of nodes). However, it is not realistic to consider that all the institutions participating to a medical grid could implement the same distributed file system, both for legacy reasons, economic reasons and security concerns. Thanks to its structure of drivers, *DM*^{2} offers the possibility to integrate in a single space of sharing data stored in a distributed file system and data stored in hospital local file systems, thus providing a global access to the data.

7.3 A Datacentric Schema

Medical Images classify in the kind of datasets proposed to be applied for a ***Datacentric Grid*** [70] (see section 3.1.2) :

- they represent large volumes of data and require much computing power.
- they are geographically distributed; medical information is collected in different health centers, hospitals, care units, even for the same patient.
- they are *(in practice)* immovable because of the confidentiality of the data and the latency for fetching data.

The diversity of image processing algorithms could make interesting the characteristics of mobile code, considering a scenario for hybrid queries where a user

[1] A highest level access to database managers.

transparently moves her/his code to the data (distributed in multiple sites) instead of collecting the data in a central place and then running the algorithm. It addresses also problems of mobile users and security.

The *Datacentric Grid* is a theory which requires deep changes in the classical computing concepts, and goes in a direction which offers a set of open issues in computer science. However, the environment of execution at each site and the problems discussed in section 3.1.2 make un-viable that option at the moment.

In this thesis we used the standard model (in its youth, yet) of Grids because there exist developed platforms which can be used today, and because the *Distributed System Engines* architecture has been designed to encompass all kind of application. However, we consider the DataCentric grid model as an interesting direction of future work for high performance medical image applications.

7.4 Images Storage

To easily manipulate medical images, we have designed a storage interface to DICOM medical servers, on top of the **DICOM3** standard [49] [50]. This proved to be difficult since DICOM data are not structured as flat files but as collections of image slices (DICOM series) and DICOM slices are containing both raw image data and metadata. This interface was developed as a ReQuest Driver (RQD) DICOM compliant by using the DICOM Tool Kit (DCMTK) ; at the moment it is operable with DICOM servers like CTN and DCMTK. Because it is DICOM compliant, it is expected to work with PACS systems [2] without any modification.

Users and Grids can take advantage of this service for obtaining access to temporal sequences of medical images (DICOM). Future work to be done, concerns the implementation of OGSA/WSRF compliant drivers.

7.5 Conclusion and Perspectives

This thesis has offered to us the opportunity to propose a framework for designing and implementing distributed systems based on local services and data servers while sharing a global data and application space through the connection to a grid middleware. We in particular introduced a new multi-layered entity, the *engines*. Hierarchically structured with respect to the semantic complexity of the implemented functions, allowing one to define and integrate a variety of communicating components, engines constitute the functional bricks on which we propose to design distributed applications.

This framework has been specially designed to allow the implementation of extensible, open and secure high performance and data intensive solutions. A first prototype of distributed medical image hybrid query system, DSEM/DM^2, has proved the feasibility of our proposals.

Future work includes :

[2] The Cardiological Hospital at Lyon, where we have done our tests, is considering to install a PACS system in replacement of the CTN DICOM Server.

- the upgrading of our prototype into an operational system usable in a hospital environment. Composed of a core middleware (*DSEM*) and a medical application (DM^2), the actual prototype constitutes a solid base to build the final system.
- the adaptation of the DSE architecture to a datacentric model. A first idea could be to interface DSE drivers with mobile agents [103]. However this will require us to study new paradigms for sharing information (e.g., metadata), enforcing the data privacy and maintaining an exhaustive and consistent view of the information space and of the active entities, managing potentially mobile users.

–

Chapitre 8

Glossary, Acronyms and Definitions

*"Cynics reckon that the Grid is merely an excuse by computer scientists to milk the political system for more research grants so they can write yet more lines of useless code.", **Economist**, June 2001*

–

8.1 DSE glossary

Special terms used in association with the *DSE*.

Architecture : Definition of all the concepts required to build a multilayer *Distributed System Engine*.

Applications (types) : Client and Server

Applications (External) : Applications that run in the same host than the engine and have a IPC communication with the engine instead of network communication.

Applications (Client) : External processes to the DSE but exist in the same local machine. They are able to issue a message to a distributed system engine, and wait for a response.

Applications (Server) : External processes to the DSE but exist in the same local machine. They are able to receive incoming messages from a distributed system engine, and produce a response. A server application is based on IPC_in / IPC_out processes which could start interaction with other processes into the DSE

Distributed System (DS) : A set of intercommunicating and cooperating virtual components which we divide between **engines** and **external tools and services** (also called machines)

Distributed System Engine (DSE) : Multi-layer architecture for designing and implementing a *Distributed System* as a set of interacting *engines*, in an environment with strong requirements of high performance.

Distributed System Engine Manager (DSEM) : Our prototype implementation of the middleware layers of the **DSE** architecture.

Distributed Medical Data Manager (DM^2) : A medical system, developed over the **Distributed System Engine Manager (DSEM)**. It corresponds also to an implementation of the application layers of the architecture (**DSE**).

Daemons (Services) : see **Services (Daemon)**

Drivers : are multi-process entities which handle different kinds of *transactions* instead of single message.

Drivers (types) : QUery Drivers, ReQuest Drivers, TasK Drivers, TOol Drivers Drivers.

Driver (QUery Drivers -QUD) : are processes in charge of managing a whole transaction (query) made up of a set of *Tasks and Requests*

Driver (ReQuest Drivers -RQD) : are processes in charge of accessing remote components such as other engines and external servers.

Driver (TasK Drivers - TKD) : are processes in charge of a specialized part of a query (task)

Driver (TOol Drivers - TOD) : are processes in charge of performing internal operations.

Drivers (Services) : see **Services (Drivers)**

DSE Drivers : see **Drivers**

Engine : A complex component, composed by a set of independent local processes, which interacts by issuing messages between them. It has connections to the external world, asuch as tools, services and other engines. It is a brick to build a DS.

Engine (Client) : A type of implemented engine for the DM^2 application. It deals with the application issues in a hospital.

Engine (DM^2) : Any engine for the DM^2 application.

Engine (Server) : A type of implemented engine for the DM^2 application. It deals with regions of hospitals.

External Applications : see **Applications (External)**

IPC_in / network_out processes (IINO) : see *Local Processes*

IPC_in processes only (IIO) : see *Local Processes*

IPC_in / IPC_out processes (IIIO) : see *Local Processes*

Machine : External elements to a DS, in which there is not choice -or we are not interested in- to make any modifications : they exist as they are, and all we can do is to interface them. They represent tools and services which an engine can use.

Machine (types) : Client and Server

Machine (Client) : Any machine in the network side, able to get in touch with an network_in / IPC_out (NIIO) process.

Machine (Server) : Any machine in the network side, which could be reached by an IPC_in / network_out (IINO) process.

Message : a string of characters.

Message Passing Kernel (MPK) : is an entity in charge of providing routing of messages between **local processes**.

Network_in/ IPC_out processes (NIIO) : see *Local Processes*

Layer 0 (Message Passing) : is A set of processes plus a **message passing kernel (MPK)**

Layer 1 (Transaction) : is a set of **drivers**

Layer 2 (Distribution) : is a set of **internal tools** and **services drivers and daemons**

Layer 3 (Application) : is a set of **services**

Layer 4 (User) : a set of **interfaces**

Layers (Application) : The higher layers of the architecture including application and user.

Layers (Middleware) : The lower layers of the architecture including message passing, transaction and distribution.

Local Processes (types) : NIIO, IINO, IIO, IIIO

Local Processes (Network_in/ IPC_out processes - NIIO) : are processes which have the ability to receive inner messages from the network.

Local Processes (IPC_in / network_out processes - IINO)) : are processes which receive messages by IPC mechanisms through the message passing kernel (MPK), and send messages to the network.

Local Processes (IPC_in processes only - IIO) : are processes which principal characteristic is receiving requests -by IPC mechanisms- from local processes.

Local Processes (IPC_in / IPC_out processes - IIIO) : They receive requirements and issue also requirements to other processes using IPC mechanisms.

Package : a set of blocks of source code, which implement basic applications of the *DM^2 System*. This code is is written as transactions (queries, tasks and requests) and drivers.

Processes : see **Local Processes**, **Special Processes**

Special Process (Client) : A process of type IPC_in / IPC_out (IIIO) which is able to issue a message into a distributed system engine, and waits for a response.

Special Process (Server) : A process of type IPC_in / IPC_out (IIIO) which is able to receive an incoming messages from a distributed system engine, and produces a response.

Special Processes (types) : Client and Server

Query : see **Transactions**

Request : see **Transactions**

Service : is the job performed by an application and *offered* to its users.

Service (External) : is the work performed by an external application, and *used* by an engine.

Service (DM^2) : A *service* offered by the DM^2 application such as hybrid queries, queries by content, or data access.

Service (DAemon - SDA) : is a set of 1 or many Query Drivers (QUD). It deals with all the problems related to the accessing one *DM^2 Service*.

Service (DRiver - SDR) : is a group of RQD in charge of solving different possibilities of a request, by accessing *external machines*.

Task : see **Transactions**

Tool (internal) : an entity having components of the three middleware layers of an engine. It is used as an instrument for performing low level works.

Tool (external) : an instrument for performing low level work, but external to an engine, e.g., a cache tool. In other words, it is an external low level service.

Transactions (types) : Query, Task, Request

Transactions (Queries) : are a set of *Tasks and Requests*

Transactions (Tasks) : are a set of Requests

Transactions (Requests) : are a set of messages to a service in the network side.

User : The one who uses the *DM^2 Application* including external applications, client machines, other engines, Grid, etc.

8.2 Acronyms

API : Application Programming Interface
ASP : Application Service Provider
ATF : DataGrid Architecture Task Force
CAE : Computer-Aided Engineering
CAN : Content-Addressable Networks (P2P)
CAS : Community Autorisation Service
CBIR : Content-Based Image Retrieval
CBVIR : Content-Based Visual Information Retrieval
CE : Computing Element (WP4)
CERT : X.509 CERTificate
ClassAd : Condor CLASSified ADvertisement language
CPU : Computing Processing Unit
CTN : Central Test Node

DAG : Directed Acyclic Graph (unicore)
DBMS : Data Base Management System
DM2 : Distributed Medical Data Manager
DICOM : Digital Image and COmmunication in Medicine
DS : Distributed System.
DSE : Distributed System Engine
EF : Ejection Fraction
FTP : File Transfer Protocol
GIS : Global Information Service
GIIS : Grid Index Information Service (globus)
GGF : Global Grid Forum
GMA : Grid Monitoring Architecture
GPCALMA : Grid Platform for Computer Assisted Library for MAmmography
GRACE : Grid Architecture for Computational Economy (NIMROD-G)
GRIS : Grid Resource Information Service (globus)
GSI : Globus Security Infrastructure
GUI : Graphical User Interface
HEP : High-Energy Physics
HIS : Hospital Information System
HPC : High Performance Computing environment
HPSS : High Performance Storage System
HTTP : HyperText Transfer Protocol
HTC : High Throughput Computing environment
HSM : Hierarchical Storage Manager
IDL : Interface Description Language (legion)
IEEE : Institute of Electrical and Electronics Engineers
IINO : Ipc_In / Network_Out processes
IIO : Ipc_In processes Only
IIIO : Ipc_In / Ipc_Out processes
IPC : Inter Process Communication
JDL : Job Description Language
JSS : Job Submision Service
QUD : QUery Driver
LAN : Local Area Network
LB : Logging and Bookkeeping
LCAS : Local Centre Autorisation Service
LCMAPS : Local Credential MAPping Service
LDAP : Lightweight Directory Access Protocol [160]
LFN : Logical File Name
LHC : Large Hadron Collider
MDS : Metacomputing Directory Services (globus)
MPI : Message Passing Interface
MPK : Message Passing Kernel
MPP : Massively Parallel Processing
MSS : Mass Storage System
MSSRM : Mass Storage System Reference Model (IEEE)

NIIO : Network_In/ Ipc_Out processes
NFS : Networked File System
OGSA : Open Grid Services Architecture
OGSI : Open Grid Services Infrastructure
PACS : Picture Archiving and Communication System
PCK : Package
PDC : Parallel and Distributed Computing
PFN : Physical File Name
PKI : Public Key Infrastructure
PVM : Parallel Virtual Machine
P2P : Peer to Peer
QoS : Quality of Service
QRSC : Query Retrieve Service Class - DICOM
RAID : Redundant Array of Inexpensive Disks
RB : Resource Broker
RC : Replica Catalog
RIS : Radiological Information Systems
RM : Replica Manager
RQD : ReQuest Driver
RSA : R. Rivest, A. Shamir, L. Adleman, Public-Key Encription System
SAN : Storage Area Network
SDA : Service DAemons
SDR : Service DRivers
SE : Storage Element
SDSC : San Diego Supercomputer Center
SMF : Standard Mammogram Form
SOA : Service Oriented Architecture
SOAP : Simple Object Access Protocol
SQL : Standard Query Language
SRB : Storage Resource Broker
SSD : UML System Sequence Diagram
SSL : Secure Sockets Layer
TCP Transport Communication Protocol
TFN : Transfer File Name
TKD : TasK Driver
TOD : TOol Driver
UI : User Interface
UML : Unified Modeling Language
UNICORE : UNiform Interface to COmputing REsources
VO : Virtual Organization
WAN : Wide Area Network
WDSL : Web Services Description Language
WMS : Workload Management System
WP : datagrid Work Package
WSRF : WS-Resource Framework
XML : eXtensible Markup Language

8.3 Definitions

Client :
An application program that establishes connections for the purpose of sending requests.

Clusters :
They are, in some sense, the predecessors of the grid technology. Clusters interconnect nodes through a local high-speed network, using commodity hardware, with the aim at reducing the costs of such infrastructures. Supercomputers have been replaced by cluster of workstations in a huge number of research projects, thus the grid technology is the natural evolution of clusters.

Content-based Query :
A query which makes additional computation into an image. It uses methods that automatically extract visual features such as colours, contours, and textures to describe and search images. Alternative search methods are usually text-based [34].

Distributed Computing :
A type of computing in which different components and objects of an application can be located on different network connected computers. Any system where many computers solve a problem together.

Engine :
The Webster's dictionary [190] defines an *engine* as "something used to achieve a purpose". For example, in databases theory, it is the part of a database management system (DBMS) that stores and retrieves data. In Web Computing, it is refereed to *search engines*, and is a general class of programs that search documents for specified keywords and return a list of the documents where the keywords were found. In these examples, the concept of *engine* is used to search for data or information. We use the term *engine* because we also search for data and information, but with additional and specific characteristics : (i) huge quantity of data, (ii) data is stored as raw data in image files and also as metadata in databases, (iii) to get information, a computing process must be done (query by content).

Grid :

Some classical definitions :
- "Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements" [159].
- "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities" [8]

- "Grid is a networked computer resource sharing and coordinated problem solving in dynamic, multi-institutional virtual organizations" [5]
- "Grids are geographically distributed, heterogeneous collections of computing resources that can be accessed through a single point of contact. They provide computational power or access to data at scales beyond even the largest single system" [174] [69] .

HTC vs HPC :
An environment which can provide [61] long period of time (weeks or months) of computing power is classified as *High Throughput Computing (HTC)*. In such environment there is a special interest in the sustained throughput and the number of completed jobs, rather than the peak performance and the wall clock time. A *High Performance Computing (HPC)* environment delivers tremendous power over a short period of time and is highly dependent on the wall clock time.

Hybrid Query :
A query which needs to access metadata databases, access raw data images, and make content-based retrieval.

Image Processing :
Analyzing and manipulating images with a computer ; it is the application of signal processing techniques to the domain of images.

Message Passing :
Message passing is a general term for a variety of strategies for structured interclient communication. The required steps to send and receive messages are specific to the technic.

Middleware :
A software layer that functions as a conversion or translation layer ; it's a consolidator and integrator which can encompass different architectures, operating systems, and physical locations. This technology allows to connect networks, workstations, supercomputers and other computer resources together into a system that can encompass different architectures, operating systems and physical locations [19]. Technologies as Grids, Proxies and MSS have complex middleware components that allow them to optimize resources usage and to offer additional services.

Picture Archiving and Communication System (PACS) :
It is an electronic system dedicated to medical image management. It enables the following image cycle : production on imaging devices, archiving, transfer via network and consultation, processing and softcopy reading on stations. PACS relies on the DICOM standard for image representation and communication services [155].

Proxy : An intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. The requests are serviced internally or by passing them, with possible translation, on to other servers. A proxy must interpret and, if necessary, rewrite a request message before forwarding it.

Server :

An application program that accepts connections in order to service requests by sending back responses.

Service :
A *Service* is an independent application, with a known interface, and which performs a work for their users. It includes data collection, conversion, and storage ; communication services ; archive services such as catalog queries and data distribution ; and control authority services such as registration and distribution certificates.

Transaction :
The execution of a program accessing shared data at multiple sites, which garanties the ACID properties : *atomic, consistent, isolated and durable.* It's a sequential program that always execute to completion [1]

Chapitre 9

Annexes

"Tout le monde savait que c'ètait impossible. Puis vint un imbècile qui ne le savait pas et qui l'a fait", ***M. Pagnol***

–

9.1 Annexe A : Machine configuration for an Engine Server at INSA Lyon

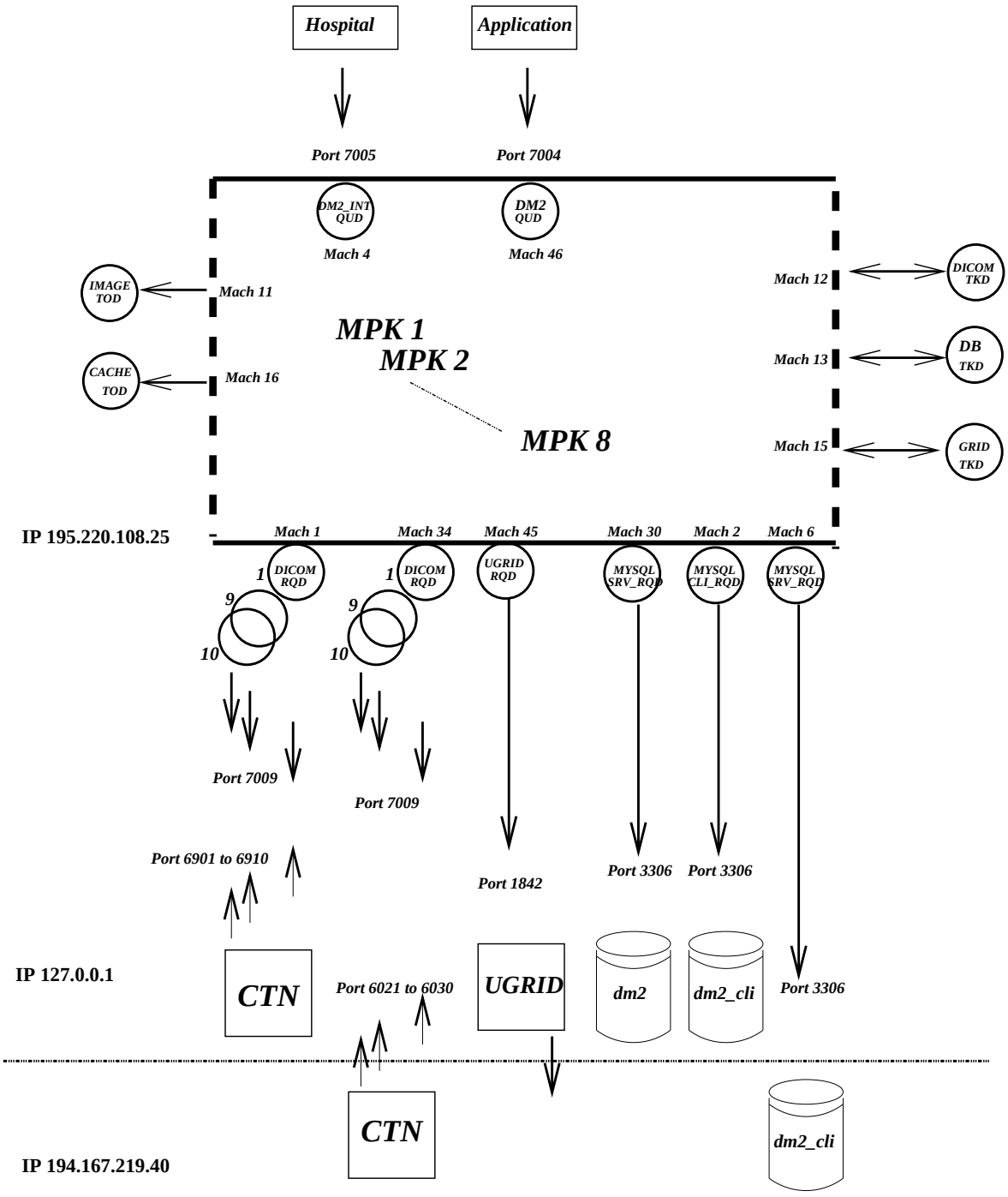


FIG. 9.1 – DM^2 Server Engine at INSA - Lyon

Comments

The figure 9.1 and the file configuration show a DM^2 Server used to test the DM^2 (medical) applications and described in chapter 6.

- the three MYSQL Request Drivers (RQD), compose a MYSQL Service Driver (SDR)
- the two DICOM Request Drivers (RQD), compose a DICOM Service Driver (SDR)
- the two DM^2 Query Drivers (QUD), compose a DM^2 Service Daemon (SDA)
- There is an Image Tool composed by only one Image Tool Driver (TOD)
- There is a Cache Tool composed by only one Cache Tool Driver (TOD). This is a test only Tool, because the real one is being developed at LIRIS [149] by another research team [187].
- Each DICOM TKD has defined ten (10) copies, always means that it can transfer (DICOM pull) up to 10 slices in parallel. It always contacts CTN at the same port (here, the port 7009), and it receives DICOM slices by different ports (i.g., 6901 to 6910, or 6021 to 6030).
- The DM^2 Server database is local, and the DM^2 Client Database is remote.
- A local CTN exists for simulating a second (local) hospital. Because of this, there exists a local DM^2 Client Database also. Remember that an engine can get in touch with many Databases.
- The DM^2 Internal QUD (DM2 INT) is the one used for getting files and data from the Hospital.
- The second, DM^2 QUD (DM2 QUD), is used to get in touch with applications that use the API layer 3.
- For improving performance, we have define 8 copies of the Message Passing Kernel (MPK).

Configuration File[1]

```
export machNR_1=1
export machACTIVE_FLAG_1=YES
export machSTART_FLAG_1=YES
export machNAME_1=HOSPITAL_gibbon
export machIN_PORT_1=6901
export machIP_1=195.220.108.25
export machIP_NAME_1=gibbon
export machPORT_1=7009
export machTYPE_1=DICOM
export machHOST_1=ANY_ONE
export machUSER_1=ANY_ONE
export machPASSWD_1=ANY_ONE
export machDB_1=ANY_ONE
export machNR_SLAVES_1=10
```

[1]based on the Bash file : DSEM504-DM2SRV.gibbon.BASH

```
export machNR_2=2
export machACTIVE_FLAG_2=YES
export machSTART_FLAG_2=YES
export machNAME_2=MYSQL_CLI_RQD_gibbon
export machIN_PORT_2=ANY_ONE
export machIP_2=195.220.108.25
export machIP_NAME_2=gibbon
export machPORT_2=3306
export machTYPE_2=MYSQL_CLI_RQD
export machHOST_2=gibbon.creatis.insa-lyon.fr
export machUSER_2=hDSEM
export machPASSWD_2=hDSEM
export machDB_2=dm2_cl
export machNR_SLAVES_2=1

export machNR_4=4
export machACTIVE_FLAG_4=YES
export machSTART_FLAG_4=YES
export machNAME_4=LYON_INTERNAL_DM2_CELL
export machIN_PORT_4=7005
export machIP_4=0.0.0.0
export machIP_NAME_4=gibbon
export machPORT_4=ANY_ONE
export machTYPE_4=DM2_INTERNAL_TXDD
export machHOST_4=ANY_ONE
export machUSER_4=ANY_ONE
export machPASSWD_4=ANY_ONE
export machDB_4=ANY_ONE
export machNR_SLAVES_4=1

export machNR_6=6
export machACTIVE_FLAG_6=YES
export machSTART_FLAG_6=YES
export machNAME_6=MYSQL_CLI_RQD_HOSPITAL_dcmtkhd
export machIN_PORT_6=ANY_ONE
export machIP_6=194.167.219.40
export machIP_NAME_6=gibbon
export machPORT_6=3306
export machTYPE_6=MYSQL_CLI_RQD
export machHOST_6=creatis-dcmtkhd.univ-lyon1.fr
export machUSER_6=hDSEM
export machPASSWD_6=hDSEM
export machDB_6=dm2_cl
export machNR_SLAVES_6=1
```

```
export machNR_11=11
export machACTIVE_FLAG_11=YES
export machSTART_FLAG_11=YES
export machNAME_11=IMAGE_TOOLS
export machIN_PORT_11=ANY_ONE
export machIP_11=127.0.0.1
export machIP_NAME_11=gibbon
export machPORT_11=ANY_ONE
export machTYPE_11=IMAGE_TOD
export machHOST_11=ANY_ONE
export machUSER_11=ANY_ONE
export machPASSWD_11=ANY_ONE
export machDB_11=ANY_ONE
export machNR_SLAVES_11=2

export machNR_12=12
export machACTIVE_FLAG_12=YES
export machSTART_FLAG_12=YES
export machNAME_12=DICOM_TASK_DRIVER
export machIN_PORT_12=ANY_ONE
export machIP_12=127.0.0.1
export machIP_NAME_12=gibbon
export machPORT_12=ANY_ONE
export machTYPE_12=DICOM_TKD
export machHOST_12=ANY_ONE
export machUSER_12=ANY_ONE
export machPASSWD_12=ANY_ONE
export machDB_12=ANY_ONE
export machNR_SLAVES_12=2

export machNR_13=13
export machACTIVE_FLAG_13=YES
export machSTART_FLAG_13=YES
export machNAME_13=DATABASE_TASK_DRIVER
export machIN_PORT_13=ANY_ONE
export machIP_13=127.0.0.1
export machIP_NAME_13=gibbon
export machPORT_13=ANY_ONE
export machTYPE_13=DATABASE_TKD
export machHOST_13=ANY_ONE
export machUSER_13=ANY_ONE
export machPASSWD_13=ANY_ONE
export machDB_13=ANY_ONE
export machNR_SLAVES_13=1

export machNR_15=15
```

```
export machACTIVE_FLAG_15=YES
export machSTART_FLAG_15=YES
export machNAME_15=GRID_TASK_DRIVER
export machIN_PORT_15=ANY_ONE
export machIP_15=127.0.0.1
export machIP_NAME_15=gibbon
export machPORT_15=ANY_ONE
export machTYPE_15=GRID_TKD
export machHOST_15=ANY_ONE
export machUSER_15=ANY_ONE
export machPASSWD_15=ANY_ONE
export machDB_15=ANY_ONE
export machNR_SLAVES_15=1

export machNR_16=16
export machACTIVE_FLAG_16=YES
export machSTART_FLAG_16=YES
export machNAME_16=CACHE_ASYNC
export machIN_PORT_16=ANY_ONE
export machIP_16=127.0.0.1
export machIP_NAME_16=gibbon
export machPORT_16=ANY_ONE
export machTYPE_16=CACHE_TEST1
export machHOST_16=ANY_ONE
export machUSER_16=ANY_ONE
export machPASSWD_16=ANY_ONE
export machDB_16=ANY_ONE
export machNR_SLAVES_16=1

export machNR_30=30
export machACTIVE_FLAG_30=YES
export machSTART_FLAG_30=YES
export machNAME_30=MYSQL_SRV_RQD
export machIN_PORT_30=ANY_ONE
export machIP_30=195.220.108.25
export machIP_NAME_30=gibbon
export machPORT_30=3306
export machTYPE_30=MYSQL_SRV_RQD
export machHOST_30=gibbon.creatis.insa-lyon.fr
export machUSER_30=hDSEM
export machPASSWD_30=hDSEM
export machDB_30=dm2
export machNR_SLAVES_30=1

export machNR_34=34
export machACTIVE_FLAG_34=YES
```

```
export machSTART_FLAG_34=YES
export machNAME_34=HOSPITAL_dcmtkhd
export machIN_PORT_34=6021
export machIP_34=194.167.219.40
export machIP_NAME_34=creatis-dcmtkhd
export machPORT_34=7009
export machTYPE_34=DICOM
export machHOST_34=ANY_ONE
export machUSER_34=ANY_ONE
export machPASSWD_34=ANY_ONE
export machDB_34=ANY_ONE
export machNR_SLAVES_34=10

export machNR_45=45
export machACTIVE_FLAG_45=YES
export machSTART_FLAG_45=YES
export machNAME_45=UGRID_farmanager
export machIN_PORT_45=ANY_ONE
export machIP_45=ANY_ONE
export machIP_NAME_45=gibbon
export machPORT_45=1842
export machTYPE_45=UGRID_RQD
export machHOST_45=gibbon.creatis.insa-lyon.fr
export machUSER_45=ANY_ONE
export machPASSWD_45=user
export machDB_45=ANY_ONE
export machNR_SLAVES_45=1

export machNR_46=46
export machACTIVE_FLAG_46=YES
export machSTART_FLAG_46=YES
export machNAME_46=DM2_SERVER_TXDD
export machIN_PORT_46=7004
export machIP_46=0.0.0.0
export machIP_NAME_46=gibbon
export machPORT_46=ANY_ONE
export machTYPE_46=DM2_SERVER_TXDD
export machHOST_46=ANY_ONE
export machUSER_46=ANY_ONE
export machPASSWD_46=ANY_ONE
export machDB_46=ANY_ONE
export machNR_SLAVES_46=1

# lastActiveMACHINE_NUMBER=last( machNR + 1 )
export lastActiveMACHINE_NUMBER=47
```

```
export mpkNR=8
```

9.2 Annexe B : Machine configuration for an Engine Client at Cardiological Hospital of Lyon

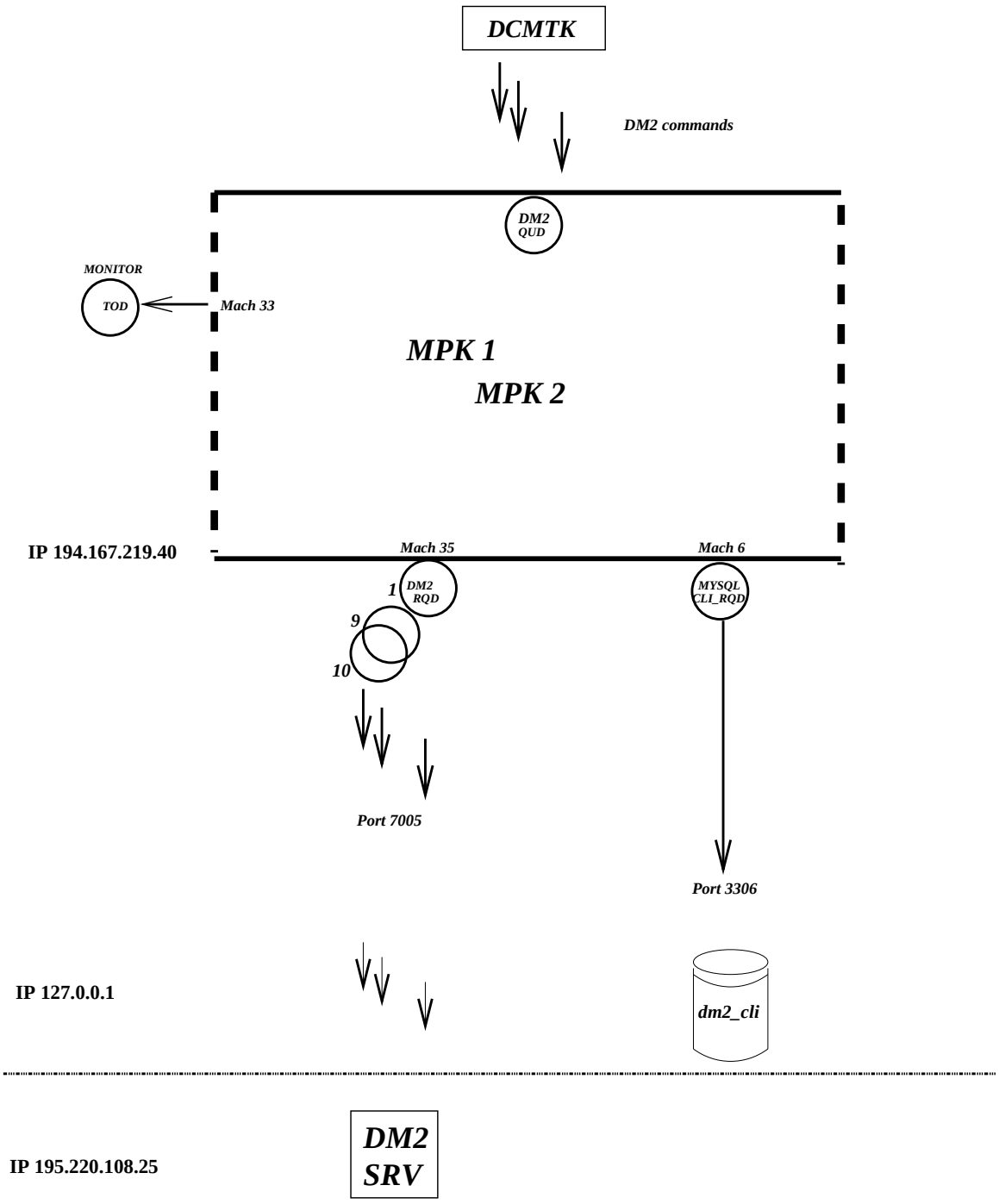


FIG. 9.2 – DM^2 Client Engine at Cardiological Hospital - Lyon

Comments

The figure 9.2 and the file configuration shows a DM^2 Client which was installed at the Cardiological Hospital of Lyon. It deals with capture of images from the IRM and DICOM scanners, as described in chapter 6.

- a low level tool driver (TOD, machine 33) was developed in order to see the behavior of the Client Engine when transferring files to the server.
- The transfer of metadata between the client engine and the server engine is done in parallel.
- We have defined only two copies of the Message Passing Kernel (MPK).

Configuration File[2]

```
export machNR_6=6
export machACTIVE_FLAG_6=YES
export machSTART_FLAG_6=YES
export machNAME_6=MYSQL_CLI_RQD_HOSPITAL_dcmtkhd
export machIN_PORT_6=ANY_ONE
export machIP_6=194.167.219.40
export machIP_NAME_6=gibbon
export machPORT_6=3306
export machTYPE_6=MYSQL_CLI_RQD
export machHOST_6=creatis-dcmtkhd.univ-lyon1.fr
export machUSER_6=hDSEM
export machPASSWD_6=hDSEM
export machDB_6=dm2_cl
export machNR_SLAVES_6=1

export machNR_17=17
export machACTIVE_FLAG_17=YES
export machSTART_FLAG_17=NO
export machNAME_17=MONITOR
export machIN_PORT_17=NO_ONE
export machIP_17=127.0.0.1
export machIP_NAME_17=creatis-dcmtkhd
export machPORT_17=NO_ONE
export machTYPE_17=hDSEM_MONITOR_TOD
export machNR_SLAVES_17=1

export machNR_33=33
export machACTIVE_FLAG_33=YES
export machSTART_FLAG_33=YES
export machNAME_33=ECHO_TOD
export machIN_PORT_33=NO_ONE
export machIP_33=127.0.0.1
```

[2]based on the Bash file : DSEM504-DM2CLI.dcmtkhd.BASH

```
export machIP_NAME_33=dhcp-4
export machPORT_33=NO_ONE
export machTYPE_33=ECHO_TOD
export machNR_SLAVES_33=1

export machNR_35=35
export machACTIVE_FLAG_35=YES
export machSTART_FLAG_35=YES
export machNAME_35=TO____REMOTE_INTERNAL_DM2_CELL
export machIN_PORT_35=NO_ONE
export machIP_35=195.220.108.25
export machIP_NAME_35=creatis-dcmtkhd
export machPORT_35=7005
export machTYPE_35=DM2_RQD
export machNR_SLAVES_35=10

# lastActiveMACHINE_NUMBER=last( machNR + 1 )
export lastActiveMACHINE_NUMBER=37

export mpkNR=2
```

9.3 Annexe C : Access to DCMTK and CTN at Cardiological Hospital of Lyon

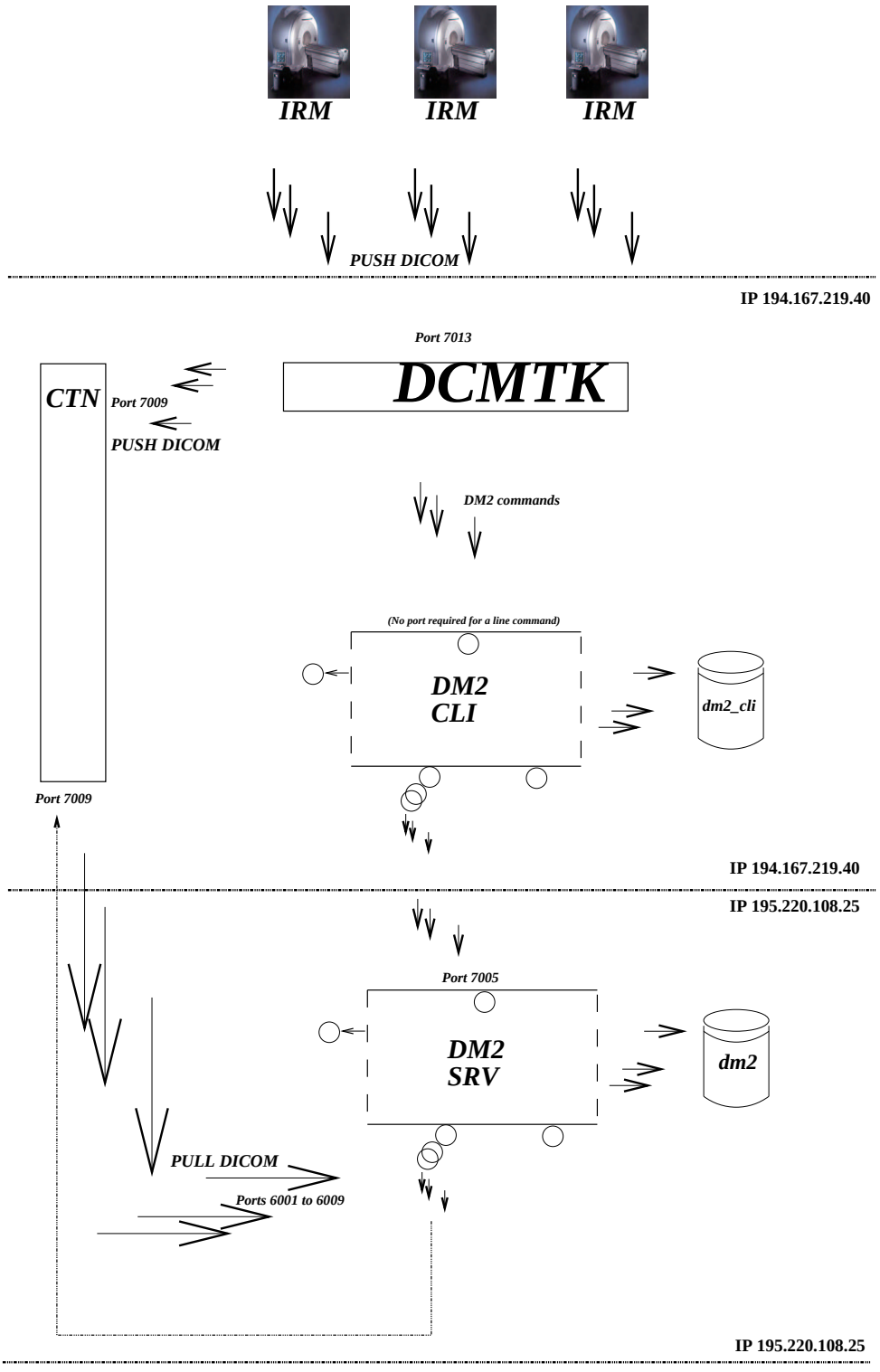


FIG. 9.3 – Access to DCMTK and CTN at Cardiological Hospital - Lyon

Comments

The figure 9.3 shows details of pushing DICOM files to a DCMTK, CTN and DM^2 at the Cardiological Hospital of Lyon.

- The DM^2 Server Engine makes PULL in parallel of the DICOM slices by using up to 10 ports. These ports must be defined in CTN and also in DM^2 (see annexe A for definition in DM^2)
- It represents only one Hospital, but it could be also seen as many hospitals, simultaneously connected to a DM^2 Server Engine.

Starts CTN and offers port 7009

```
/opt/ctn/bin/archive_server 7009 >>/tmp/ctn.log 2>&1&
```

Starts DCMTK and offers port 7013

```
$DCMTK_HOME/storescp -xcr "${DM2_HOME}/bin/storeSCP_SEQ_file #p #f"
-xcs "/usr/local/hDSEM/bin/storeSCP_ANALIZE_SEQ_BY_SERIES #p -algo 003
-last_put" -ss "irm_7013/img" -tos 5 7013&
```

9.4 Annexe D : Machine configuration for a High Performance Engine Server at INSA of Lyon

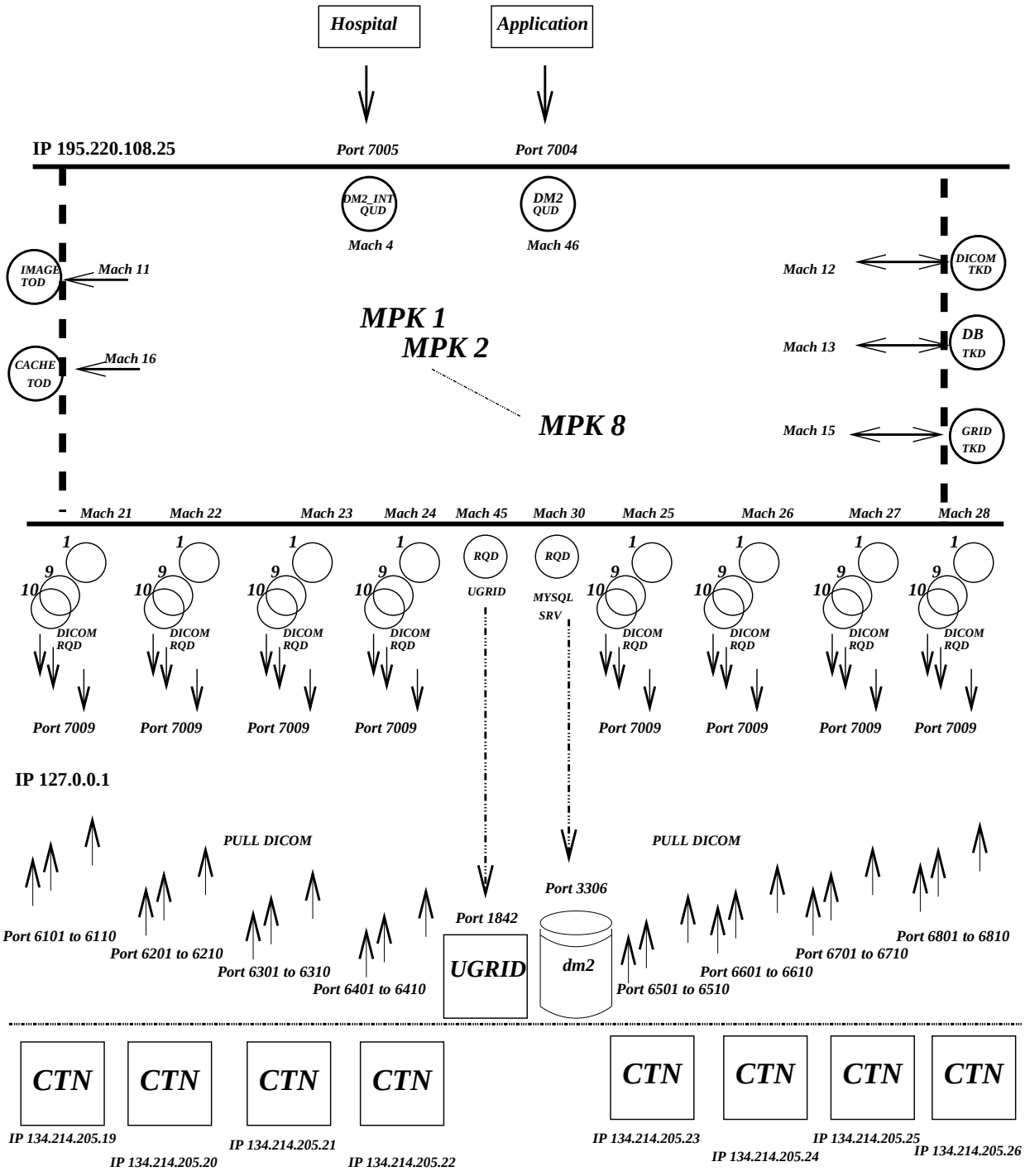


FIG. 9.4 – DM^2 High Performance Server Engine at INSA - Lyon

Comments

The figure 9.4 and the file configuration show a DM^2 High Performance Engine which was installed at Creatis INSA of Lyon for testing stressing conditions in the engine. The experiments consider parallel access over 8 CTN remote servers which represent hospitals attached to the same DM^2 Server Engine. The experiment is described in chapter 5.

- The configuration for TOD (machines 11, 16), TKD (machines 12, 13, 15), QUD (machines 4, 46), and RQD (machines 30, 45), is the same as it was showed in annexe A. The configuration file below includes the additional machines (21 to 28).

Hardware and Software

We used a Cluster of **8 PC** with processor speed of **1GHz** per processor, and memory of **1GB**. In the server engine, there were installed hard disks **udma5 ATA** (locally) and **IDE RAID** (remotely). The network speed was **100Mbits**. The cluster was running with :

```
Linux RedHat 7.3
CTN version 2.11.0
DCMTK version 3.52
PVM version 3.4.4
```

Complement Configuration File

```
export machNR_21=21
export machACTIVE_FLAG_21=YES
export machSTART_FLAG_21=YES
export machNAME_21=HOSPITAL_frodo
export machIN_PORT_21=6101
export machIP_21=134.214.205.19
export machIP_NAME_21=frodo
export machPORT_21=7009
export machTYPE_21=DICOM
export machHOST_21=ANY_ONE
export machUSER_21=ANY_ONE
export machPASSWD_21=ANY_ONE
export machDB_21=ANY_ONE
export machNR_SLAVES_21=10

export machNR_22=22
export machACTIVE_FLAG_22=YES
export machSTART_FLAG_22=YES
export machNAME_22=HOSPITAL_merry
```

```
export machIN_PORT_22=6201
export machIP_22=134.214.205.20
export machIP_NAME_22=merry
export machPORT_22=7009
export machTYPE_22=DICOM
export machHOST_22=ANY_ONE
export machUSER_22=ANY_ONE
export machPASSWD_22=ANY_ONE
export machDB_22=ANY_ONE
export machNR_SLAVES_22=10

export machNR_23=23
export machACTIVE_FLAG_23=YES
export machSTART_FLAG_23=YES
export machNAME_23=HOSPITAL_pippin
export machIN_PORT_23=6301
export machIP_23=134.214.205.21
export machIP_NAME_23=pippin
export machPORT_23=7009
export machTYPE_23=DICOM
export machHOST_23=ANY_ONE
export machUSER_23=ANY_ONE
export machPASSWD_23=ANY_ONE
export machDB_23=ANY_ONE
export machNR_SLAVES_23=10

export machNR_24=24
export machACTIVE_FLAG_24=YES
export machSTART_FLAG_24=YES
export machNAME_24=HOSPITAL_sam
export machIN_PORT_24=6401
export machIP_24=134.214.205.22
export machIP_NAME_24=sam
export machPORT_24=7009
export machTYPE_24=DICOM
export machHOST_24=ANY_ONE
export machUSER_24=ANY_ONE
export machPASSWD_24=ANY_ONE
export machDB_24=ANY_ONE
export machNR_SLAVES_24=10

export machNR_25=25
export machACTIVE_FLAG_25=YES
export machSTART_FLAG_25=YES
export machNAME_25=HOSPITAL_legolas
export machIN_PORT_25=6501
```

```
export machIP_25=134.214.205.23
export machIP_NAME_25=legolas
export machPORT_25=7009
export machTYPE_25=DICOM
export machHOST_25=ANY_ONE
export machUSER_25=ANY_ONE
export machPASSWD_25=ANY_ONE
export machDB_25=ANY_ONE
export machNR_SLAVES_25=10

export machNR_26=26
export machACTIVE_FLAG_26=YES
export machSTART_FLAG_26=YES
export machNAME_26=HOSPITAL_gimli
export machIN_PORT_26=6601
export machIP_26=134.214.205.24
export machIP_NAME_26=gimli
export machPORT_26=7009
export machTYPE_26=DICOM
export machHOST_26=ANY_ONE
export machUSER_26=ANY_ONE
export machPASSWD_26=ANY_ONE
export machDB_26=ANY_ONE
export machNR_SLAVES_26=10

export machNR_27=27
export machACTIVE_FLAG_27=YES
export machSTART_FLAG_27=YES
export machNAME_27=HOSPITAL_boromir
export machIN_PORT_27=6701
export machIP_27=134.214.205.25
export machIP_NAME_27=boromir
export machPORT_27=7009
export machTYPE_27=DICOM
export machHOST_27=ANY_ONE
export machUSER_27=ANY_ONE
export machPASSWD_27=ANY_ONE
export machDB_27=ANY_ONE
export machNR_SLAVES_27=10

export machNR_28=28
export machACTIVE_FLAG_28=YES
export machSTART_FLAG_28=YES
export machNAME_28=HOSPITAL_aragorn
export machIN_PORT_28=6801
export machIP_28=134.214.205.26
```

```
export machIP_NAME_28=aragorn
export machPORT_28=7009
export machTYPE_28=DICOM
export machHOST_28=ANY_ONE
export machUSER_28=ANY_ONE
export machPASSWD_28=ANY_ONE
export machDB_28=ANY_ONE
export machNR_SLAVES_28=10

export mpkNR=8
```

9.5 Annexe E : Server Database Description

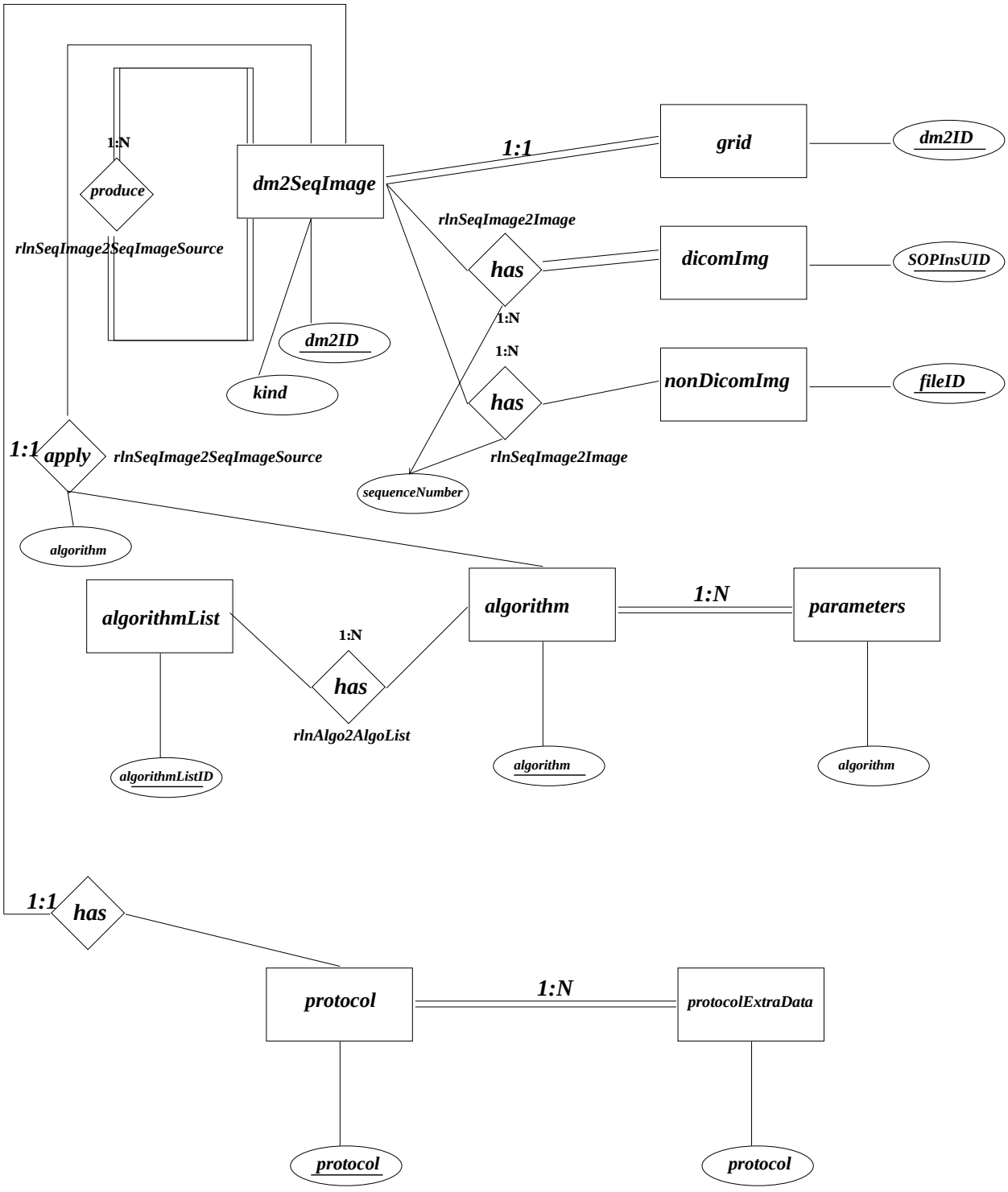


FIG. 9.5 – DM^2 Server Database : entity-relationship diagrams [97] [96]

```
mysql> show tables;
+----------------------------+
| Tables_in_dm2              |
+----------------------------+
| algorithm                  |
| algorithmList              |
| dicomImg                   |
| dm2SeqImage                |
| grid                       |
| nonDicomImg                |
| parameters                 |
| protocol                   |
| protocolExtraData          |
| rlnAlgo2AlgoList           |
| rlnSeqImage2Image          |
| rlnSeqImage2SeqImageSource |
+----------------------------+
12 rows in set (0.00 sec)
```

```
| algorithm | CREATE TABLE `algorithm` (
  `algorithm` char(64) NOT NULL default '',
  `algorithmName` char(128) default NULL,
  PRIMARY KEY  (`algorithm`)
) TYPE=MyISAM |
```

```
| algorithmList | CREATE TABLE `algorithmList` (
  `algorithmListID` char(64) NOT NULL default '',
  `algorithmListName` char(128) default NULL,
  PRIMARY KEY  (`algorithmListID`)
) TYPE=MyISAM |
```

```
| dicomImg | CREATE TABLE `dicomImg` (
  `SOPInsUID` char(64) NOT NULL default '',
  `SerInsUID` char(64) default NULL,
  `StuInsUID` char(64) default NULL,
  `fileSize` int(10) unsigned default NULL,
  `insertDate` date default NULL,
  `insertTime` time default NULL,
  `Mod` char(16) default NULL,
  PRIMARY KEY  (`SOPInsUID`)
) TYPE=MyISAM |
```

```
| dm2SeqImage | CREATE TABLE `dm2SeqImage` (
  `dm2ID` varchar(64) NOT NULL default '',
  `kind` enum('DICOM','NODICOM') default NULL,
  `nx` int(10) unsigned default NULL,
  `ny` int(10) unsigned default NULL,
  `nz` int(10) unsigned default NULL,
  `nt` int(10) unsigned default NULL,
  `sx` float default NULL,
  `sy` float default NULL,
  `sz` float default NULL,
  `st` float default NULL,
  `vdim` int(10) unsigned default NULL,
  `type` enum('8bits','u8bits','16bits','u16bits','32bits',
'u32bits','float','double') default NULL,
  `insertDate` date default NULL,
  `insertTime` time default NULL,
  `hostName` varchar(64) NOT NULL default '',
  `userID` varchar(64) NOT NULL default '',
  `imageSize` int(10) unsigned default NULL,
  PRIMARY KEY  (`dm2ID`)
) TYPE=MyISAM |


| grid  | CREATE TABLE `grid` (
  `gridFile` varchar(255) NOT NULL default '',
  `dm2ID` varchar(64) NOT NULL default '',
  `fileKind` enum('DCM','PNG','JPG','INR','VOL') default NULL,
  PRIMARY KEY  (`gridFile`,`dm2ID`)
) TYPE=MyISAM |


| nonDicomImg | CREATE TABLE `nonDicomImg` (
  `fileID` char(64) NOT NULL default '',
  `fileName` char(255) NOT NULL default '',
  `fileSize` int(10) unsigned default NULL,
  `insertDate` date default NULL,
  `insertTime` time default NULL,
  PRIMARY KEY  (`fileID`)
) TYPE=MyISAM |


| parameters | CREATE TABLE `parameters` (
  `algorithm` char(64) NOT NULL default '',
  `parameterNumber` int(4) default NULL,
```

```
  `parameter` char(64) NOT NULL default ''
) TYPE=MyISAM |


| protocol | CREATE TABLE `protocol` (
  `protocol` varchar(64) NOT NULL default '',
  `protocolName` varchar(128) default NULL,
  `dm2IDSource` varchar(64) NOT NULL default '',
  PRIMARY KEY  (`protocol`)
) TYPE=MyISAM |


| protocolExtraData | CREATE TABLE `protocolExtraData` (
  `protocol` varchar(64) NOT NULL default '',
  `tableName` varchar(255) default NULL,
  `externalDB` varchar(16) NOT NULL default '',
  `metaDataFileName` varchar(255) NOT NULL default ''
) TYPE=MyISAM |


| rlnAlgo2AlgoList | CREATE TABLE `rlnAlgo2AlgoList` (
  `algorithmListID` char(64) NOT NULL default '',
  `algorithm` char(64) NOT NULL default ''
) TYPE=MyISAM |


| rlnSeqImage2Image | CREATE TABLE `rlnSeqImage2Image` (
  `dm2ID` char(64) NOT NULL default '',
  `fileID` char(64) NOT NULL default '',
  `imageSequenceNumber` int(4) unsigned default NULL,
  PRIMARY KEY  (`dm2ID`,`fileID`)
) TYPE=MyISAM |


| rlnSeqImage2SeqImageSource | CREATE TABLE `rlnSeqImage2SeqImageSource` (
  `dm2ID` char(64) NOT NULL default '',
  `dm2IDSource` char(64) NOT NULL default '',
  `algorithm` char(64) NOT NULL default '',
  PRIMARY KEY  (`dm2ID`,`dm2IDSource`)
) TYPE=MyISAM |
```

9.6 Annexe F : Client Database Description

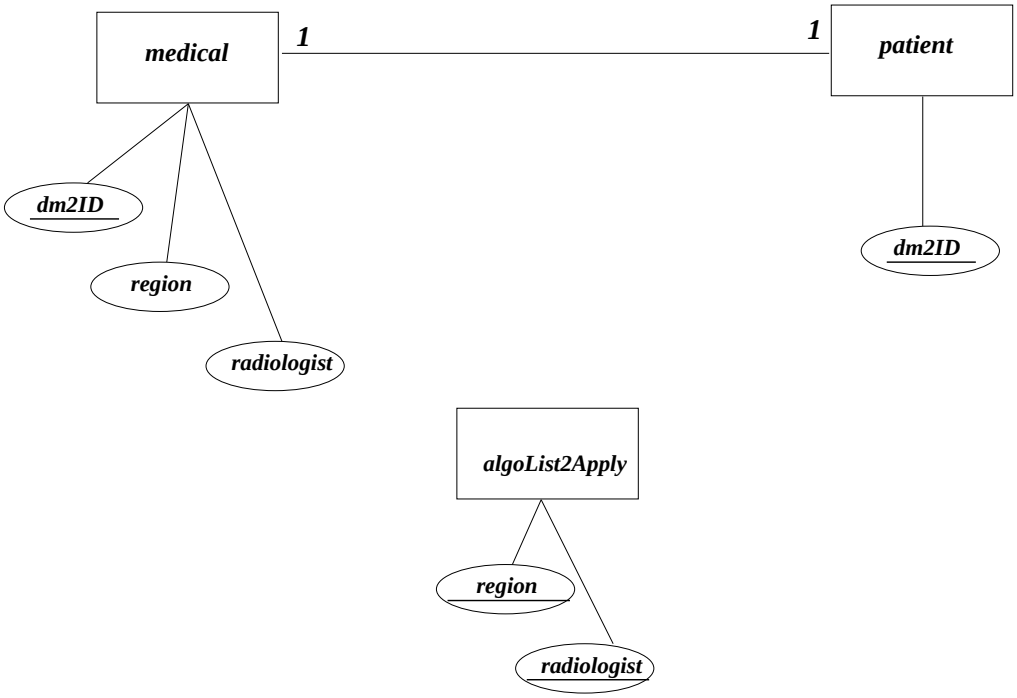


FIG. 9.6 – DM^2 Client Database : entity-relationship diagrams [97] [96]

```
mysql> show tables;
+------------------+
| Tables_in_dm2_cl |
+------------------+
| algoList2Apply   |
| medical          |
| patient          |
+------------------+
3 rows in set (0.00 sec)
```

```
| algoList2Apply | CREATE TABLE `algoList2Apply` (
  `region` enum('HEAD','THORAX','ABDOMEN','COEUR') NOT NULL default 'HEAD',
  `radiologist` varchar(128) NOT NULL default '',
  `algorithmListID` varchar(64) NOT NULL default '',
  PRIMARY KEY  (`region`,`radiologist`)
) TYPE=MyISAM |
```

```
| medical | CREATE TABLE `medical` (
```

```
  `dm2ID` varchar(64) NOT NULL default '',
  `Mod` enum('MR','US','CT','PET','SPECT','US','OTHER') default NULL,
  `region` enum('HEAD','THORAX','ABDOMEN','COEUR','OTHER') default NULL,
  `hospital` varchar(128) default NULL,
  `department` varchar(128) default NULL,
  `radiologist` varchar(128) default NULL,
  `insertDate` date default NULL,
  `insertTime` time default NULL,
  `imager` varchar(128) default NULL,
  `parameters` varchar(255) default NULL,
  `annotation` blob,
  PRIMARY KEY  (`dm2ID`)
) TYPE=MyISAM |


| patient | CREATE TABLE `patient` (
  `dm2ID` char(64) NOT NULL default '',
  `PatID` char(64) default NULL,
  `name` char(128) default NULL,
  `sexe` enum('M','F') default NULL,
  `dob` date default NULL,
  PRIMARY KEY  (`dm2ID`)
) TYPE=MyISAM |
```

9.7 Annexe G : Links to the Documentation

The author :

Hector DUQUE
http://hectorduque.free.fr
duque@creatis.insa-lyon.fr

The team :

http://hectorduque.free.fr/recherche/tdTeam.html

The API0 documentation :

http://hectorduque.free.fr/recherche/htmlAPI0/index.html

The API3 documentation :

http://hectorduque.free.fr/recherche/htmlAPI3/index.html

All the source code for DSEM/DM^2 :

http://hduque.free.fr/hDSEM/

Papers :

http://hectorduque.free.fr/recherche/tdPapers.html

This thesis document :

http://hectorduque.free.fr/thesis/

Chapitre 10

Application's Annexes

"The Grid is a solution looking for a problem", ***Jennifer M. Schopf****, Argonne National Lab, USA, 2002*

–

10.1 Annexe I : Similarity

Several similarity measurements have been implemented in our application. Let I represents the source image and J represents one target image with the same support (both I and J have n voxels). Let i denotes the gray level intensity of voxels in image I and j denotes the the gray level intensity of voxels in image J. n_i (resp. n_j) is the number of voxels with intensity i (resp. j) in image I (resp. J). n_{ij} is the number of voxels having simultaneously intensity i in image I and j in image J. $p_i = \frac{n_i}{n}$ and $p_j = \frac{n_j}{n}$ are associated probabilities. $p_{ij} = \frac{n_{ij}}{n}$ is the joint probability of a voxel at a given location to have intensity i in I and j in J. The mean and variance of intensities in image I can be computed as : $m_I = \sum_i i p_i$ and $\sigma_I^2 = \sum_i (i - m_I)^2 p_i$. From these statistical measurements, one can compute several similarity measures :

- **The simple differences :**

$$D_1(I,J) = \sum_i \sum_j p_{ij}|i-j| \text{ and } D_2(I,J) = \sum_i \sum_j p_{ij}(i-j)^2 \tag{10.1}$$

They are simple measurements for mono-modal image comparisons. They are sensitive to signal noise and inhomogeneities so their principal interest is their simplicity.

- **The coefficient of correlation :**

$$\rho^2(I,J) = \frac{\mathrm{Cov}^2(I,J)}{\mathrm{Var}(I)\mathrm{Var}(J)} = \sum_i \sum_j \frac{(i-m_I)(j-m_J)}{\sqrt{\sigma_I}\sqrt{\sigma_J}} \tag{10.2}$$

it is a normalized measurement taking into account an affine transformation between I and J intensities. It has been extensively used in the literature.

- **The Wood's criterion :**

$$\mathrm{W}(I,J) = \sum_j \frac{\sigma_{I|j}}{m_{I|j}} p_j \text{ with } \begin{cases} m_{I|j} = \frac{1}{p_i} \sum_j j p_{ij} \\ \sigma^2_{I|j} = \frac{1}{p_i} \sum_j (j - m_{J|i})^2 p_{ij} \end{cases} \tag{10.3}$$

it was introduced to register MRI on PET images. Given the set of voxels with intensity i in the source image, the Wood's criterion measures the variation of intensities of corresponding voxels in the target image.

– **The correlation ratio :**

$$\mu^2(I,J) = 1 - \frac{1}{\sigma_I^2}\sum_j p_j \sigma_{I|j}^2 \tag{10.4}$$

it is used for multi-modal registration and makes the hypothesis that a functional relation exists between the source and the target image intensities.

– **The mutual information**, or entropy :

$$H(I,J) = -\sum_i \sum_j p_{ij} \frac{p_{ij}}{p_j} \tag{10.5}$$

it is the most general similarity measure. It measures the entropy of the joint gray levels distribution without any assumption on an existing relation between source and target image intensities.

Acknowledgments

This annexe was entirely taken from the paper[44] :

J. Montagnat and H. Duque and J-M Pierson and V. Breton and L. Brunie and I.E. Magnin, ***Medical Image Content-Based Queries using the Grid****, HealthGrid, 2002*

10.2 Annexe II : 3D+time Segmentation of Magnetic Resonance Cardiac images

Accurate analysis of the cardiac function relies on tomographic image acquisitions. Current cardiac examination in MRI comprises dynamic short axis acquisitions at several slices that cover the whole heart.

The analysis of the cardiac function relies on quantitative global parameters such as the volume evolution of the left ventricular cavity, the ejection fraction, as well as local parameters such as the wall thickening. The estimation of these parameters requires the extraction of the heart contours which is a very tedious and user dependent task. Therefore, computer-assisted image processing methods are required. As in the previous example, the goal of a grid-enabled method is quasi-real time processing, in order to make the tool usable in a clinical context .

In order to ease the contour extraction process from the images, the so-called deformable model approach uses an *a priori* model of the object to be extracted. This model is deformed iteratively to fit the image content. The proposed model presents the advantage of allowing the simultaneous extraction of both the endocardial and epicardial surfaces [117, 118]. The concept, named elastic deformable template, combines a topological and geometric model of the object to be segmented, and a constitutive equation (linear elasticity) defining its behavior under applied external image forces that pushes the model interfaces towards the image edges. In this context, the *a priori* model is a bi-cavity geometrical mesh that results from the manual segmentation of a reference data set.

The equilibrium of the model is obtained through the minimization of a global energy functional : $E = E_{elastic} + E_{data}$ where $E_{elastic}$ represents the deformation energy of the model and E_{data} is the energy due to the external image forces.

Internal Energy term The object is considered as a linear elastic body. Its elastic energy can be expressed as :

$$E_{elastic} = \frac{1}{2} \int_{\Omega} \sigma^t \epsilon \; d\Omega \tag{10.6}$$

Where σ, ϵ are the 3D strain and deformation vectors, respectively and Ω is the model domain. Moreover, the material is considered as isotropic and is completely defined by the Young modulus Y and the Poisson coefficient ν. Then, with the small displacement assumption,

$$E_{elastic}(u) = \frac{1}{2} \int_{\Omega} (\mathbf{S}u)^t \mathbf{D} (\mathbf{S}u) \; d\Omega \tag{10.7}$$

where $\mathbf{S}$ is a differential operator, $\mathbf{D}(Y, \nu)$ is the elasticity matrix and u the displacement vector.

External Image Energy The object is submitted to a 3D boundary force field $\mathbf{t}$. The expression of the external energy E_{data} is :

$$E_{data}(u) = \int_{\Gamma} (\mathbf{t}.u) \; d\Gamma \tag{10.8}$$

with Γ the border of the object domain Ω. The force field is either derived from the gradient of a potential function P (such that $\mathbf{t} = -grad(P)$) computed from an edge map, or a specific force field called gradient vector flow (GVF) [119] which is sometimes more efficient regarding the initialization and the convergence to edges.

Global energy minimization The model is decomposed into tetrahedral elements. The segmentation is obtained through the minimization of the global energy functional E :

$$E(u) = \frac{1}{2}\int_{\Omega}(\mathbf{S}u)^t\mathbf{D}(\mathbf{S}u)\ d\Omega - c_F\int_{\Gamma}(\mathbf{t}.u)\ d\Gamma \tag{10.9}$$

which is achieved using the Finite Element Method (FEM) [120].

Template initialization The segmentation procedure requires a rough positioning of the initial template into the data. This is performed by an affine registration of the model with a Volume of Interest (VOI), centered on the heart region and extracted from an isotropic interpolated volume data.

The geometrical mesh is typically composed of about 6000 nodes and 25000 cells. The system handled for the problem solving is therefore of size 324MFloats ($[3 \times \text{NumberOfNodes}^2$). The program uses the Petsc library for linear algebra operations. Once the segmentation has been performed on the first frame of the sequence, the resulting model is used as the initial template for the next frame. If the sequence has 30 frames, then segmentation is repeated 30 times.

Acknowledgments

This annexe was taken from the thesis [95] paper[116] described below :

V. Breton, P. Clarysse, Y. Germain, E. Jannot, Y. Legre, C. Loomis, J. Montagnat, J-M Moureaux, A. Osorio, X. Pennec and R. Texier, ***Grid-enabling medical images analysis****, 2005*

Chapitre 11

Bibliography

11.1 References

[1] S. Mullender, ed., *Distributed Systems.* New York : Addison Wesley, pp. 18-22, 1993.

[2] Ion. Stoika and Robert. Morris and David. Liben-Nowell and David. Karger and M. Frans. Kaashoek, "Chord : A scalable peer-to-peer lookup service for internet applications," tech. rep., MIT Laboratory for Computer Science, pp. 1-12, january 2002.

[3] Frank. Dabek and Emma. Brunskill and M. Frans. Kaashoek and David. Karger and Robert. Morris and Ion. Stoika, "Building peer-to-peer systems with chord, a distributed lookup service," in *8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pp. 1–6, may 2001.

[4] Karl Czajkowski and Ian Foster and Nick Karonis and Carl Kesselman and Stuart Martin and Warren Smith and Steven Tuecke, "A resource management architecture for metacomputing systems," *Lecture Notes in Computer Science*, vol. 1459, p. 62, 1998.

[5] Foster, I. and Kesselman, C. and Tuecke, S., "The Anatomy of the Grid : Enabling Scalable Virtual Organizations," *International Journal of Supercomputer Applications*, vol. 15, no. 3, pp. 1–25, 2001.

[6] Chervenak, A. and Foster, I. and Kesselman, C. and Salisbury, C. and Tuecke, S., "The data grid : Towards an architecture for the distributed management and analysis of large scientific datasets," *Journal of Network and Computer Applications*, vol. 23, pp. 187–200, July 2000.

[7] Stockinger, H. and Samar, A. and Allcock, B. and Foster, I. and Holtman, K. and Tierney, B., "File and object replication in data grids," in *10th IEEE Symposium on High Performance and Distributed Computing (HPDC2001)*, (San Francisco, California, USA), aug 2001.

[8] Ian. Foster and Carl. Kesselman, *The Grid : Blueprint for a New Computing Infrastructure.* San Fransisco, CA, USA : Morgan-Kaufmann Publishers, Inc, pp. 157-177, July 1998.

[9] Ian. Foster and Carl. Kesselman, *The Grid 2 : Blueprint for a New Computing Infrastructure*. San Fransisco, CA, USA : Morgan-Kaufmann Publishers, Inc, 2 ed., 2003.

[10] Ian. Foster and Carl. Kesselman and G. Tsudik and S. Tuecke, "A security architecture for computational grids," in *Fith ACM Conference on computers and Communications Security*, (New York), november 1998.

[11] I. Foster and C. Kesselman and J. Nick and S. Tuecke", "The physiology of the grid : An open grid services architecture for distributed systems integration," 2002.

[12] Ian Foster and Carl Kesselman, *The Grid : Blueprint for a New Computing Infrastructure*. San Fransisco, CA, USA : Morgan Kaufmann, July 1998.

[13] Francesco Giacomini and Francesco Prelz and Massimo Sgaravatto and Igor Terekhov and Gabriele Garzoglio and Todd Tannenbaum, "Planning on the grid : A status report. ppdg-20, particle physics data grid collaboration.," October 2002.

[14] Ian Foster and Carl Kesselman, "Globus : A metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, pp. 115–128, Summer 1997.

[15] Andrew S. Grimshaw and William A. Wulf and James C. French and Alfred C. Weaver and Paul Reynolds, "Legion : The next logical step toward a nationwide virtual computer," tech. rep., University of Virginia, 1994.

[16] Rajkumar Buyya, *Economic based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Monash University, Melbourne, Australia, April 12 2002.

[17] I. Foster and J. Geisler and S. Tuecke, "MPI on the I-WAY : A wide-area, multimethod implementation of the Message Passing Interface," pp. 10–17, IEEE Computer Society Press, 1996.

[18] Ian Foster and Adriana Iamnitchi, "On Death, taxes and the convergence of Peer-to-Peer and Grid Computing," in *2nd International Workshop on Peer-to-Peer Systems, IPTPS03*, (Berkeley, CA, USA.), NSF, 20-21 February 2003.

[19] DataGrid, "Data access and file systems : State of the art report," february 8th 2002. DataGrid WP2.

[20] DataGrid, "Final report," february 10th 2004. DataGrid WP2.

[21] DataGrid, "Final report including report on the second bio-testbed release," january 27th 2004. DataGrid WP10.

[22] DataGrid, "Review of current technologies," february 2002. DataGrid WP5.

[23] Acharya, R. and Wasserman, R. and Sevens, J. and Hinojosa, C., "Biomedical Imaging Modalities : a Tutorial," *Comput Med Imaging Graph (cmig)*, vol. 19, no. 1, pp. 3–25, 1995.

[24] Breton, V. and Medina, R. and Montagnat, J., "DataGrid, Prototype of a Biomedical Grid," *Methods MIMST*, vol. 42, no. 2, 2003. Also available as `http://www.methods-online.com/zs/startz.asp`.

[25] Huang, H. K., *PACS : Picture Archiving and Communication Systems in Biomedical Imaging*. Berlin : Springer, 1996.

[26] Montagnat, J. and Davila, E. and Magnin, I.E., "3D objects visualization for remote interactive medical applications," in *3D Data Processing, Visualization, Transmission*, (Padova, Italy), jun 2002.

[27] Roche, A. and Malandain, G. and Pennec, X. and Ayache, N., "The Correlation Ratio as a New Similarity Measure for Multimodal Image Registration," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, vol. 1496 of *LNCS*, pp. 1115–1124, oct 1998.

[28] Penney, G.P. and Weese, J. and Little, J.A. and Desmedt, P. and Hill, D.LG. and Hawkes, D.J., "A Comparison of Similarity Measures for Use in 2D-3D Medical Image Registration," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, vol. 1496 of *LNCS*, pp. 1153–1161, oct 1998.

[29] Richard McClatchey and Predrag Buncic and David Manset and Tamas Hauer and Florida Estrella and Pablo Saiz and Dmitri Rogulin, "The mammogrid project grids architecture," in *Computing in High Energy and Nuclear Physics (CHEP03)*, (La Jolla, CA, USA), p. 6 pages, Marc 2003.

[30] Cecile Germain-Renaud and Romain Texier and Angel Osorio, "Interactive exploration of medical images on the grid," in *HealthGrid*, (Clermont-Ferrand), HealthGrid, january 29th - 30th 2004.

[31] Sharon Lloyd and Marina Jirotka and Andrew Simpson and David Gavaghan and Ralph Highnam and David Watson and Mike Brady, "Digital mammography : A world without film," in *HealthGrid*, (Clermont-Ferrand), HealthGrid, january 29th - 30th 2004.

[32] Richard McClatchey and Florida Estrella and Chiara del Frate and Tamas Hauer, "Resolving clinicians queries across a grids infrastructure," in *HealthGrid*, (Clermont-Ferrand), HealthGrid, january 29th - 30th 2004.

[33] J.M. Alonso and V. Hernandez and G. Molto, "High performance cardiac tissue electrical activity simulation on a parallel environment," in *HealthGrid*, (Clermont-Ferrand), HealthGrid, january 29th - 30th 2004.

[34] Henning Muller and Arnaud Garcia and Jean-Paul Valle and Antoine Geissbuhler, "Grid computing at the university hospitals of geneva," in *HealthGrid*, (Lyon), HealthGrid, january 16th - 17th 2003.

[35] G. Geist and J. Kohl and P. Papadopoulos, "PVM and MPI : a Comparison of Features," *Calculateurs Paralleles*, vol. 8, no. 2, pp. 137–150, 1996.

[36] Al. Geist and Adam. Beguelin and Jack.Dongarra and Weicheng. JIANG and Robert. MANCHEK and Vaidy. SUNDERAM, *PVM : Parallel Virtual Machine ; Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994.

[37] Brian Barrett and Jeff Squyres and Andrew Lumsdaine, "Integration of the LAM/MPI environment and the PBS scheduling system," in *Proceedings, 17th Annual International Symposium on High Performance Computing Systems and Applications*, (Quebec, Canada), May 2003.

[38] Sriram Sankaran and Jeffrey M. Squyres and Brian Barrett and Andrew Lumsdaine and Jason Duell and Paul Hargrove and Eric Roman, "The LAM/MPI checkpoint/restart framework : System-initiated checkpointing," in *LACSI Symposium*, (Sante Fe, New Mexico, USA), Oct. 2003.

[39] Jeffrey M. Squyres and Brian Barrett and Andrew Lumsdaine, "The system services interface (SSI) to LAM/MPI," Technical Report TR575, Indiana University, Computer Science Department, 2003.

[40] B. Tierney and R. Aydt and D. Gunter and W. Smith and V. Taylor and R. Wolsky and M. Swany, "A grid monitoring architecture. technical report. gwd-perf-16-2, global grid forum." `citeseer.nj.nec.com/article/tierney02grid.html`, January 2002.

[41] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application-level scheduling on distributed heterogeneous networks," in *CD-ROM Proceedings of Supercomputing'96*, (Pittsburgh, PA), IEEE, Nov. 1996.

[42] Butler, R. and Engert, D. and Foster, I. and Kesselman, C. and Tuecke, S. and Volmer, J. and Welch V., "A National-Scale Authentication Infrastructure," *IEEE Computer*, vol. 33, no. 12, pp. 60–66, 2000.

[43] MPI Forum, "Mpi : A message passing interface standard," *International Journal of Supercomputer Applications*, vol. 8, pp. 165–416, 1994.

[44] J. Montagnat and H. Duque and J-M Pierson and V. Breton and L. Brunie and I.E. Magnin, "Medical image content-based queries using the grid," in *HealthGrid*, (Lyon, France), January 16-17 2003.

[45] Duque, Hector and Montagnat, Johan and Pierson, Jean-Marc and Magnin, Isabelle and Brunie, Lionel, "DM^2 : A Distributed Medical Data Manager for Grids," in *Biogrid 03, Tokyo May 12th to 15th 2003, proceedings of the IEEE CCGrid03.*

[46] Jean-Marc Pierson and Lionel Brunie and David Coquil, "Semantic collaborative web caching," in *WISE2002 : Web Information System Engineering*, (Singapour), pp. 30–39, dec 2002.

[47] Lionel Brunie and David Coquil and Serge Simon, "Software architectures for collaborative proxies in wide area information systems (position paper)," in *Fourth International Workshop on Parallel and Distributed Databases : innovative applications and new architectures (PaDD'2001)*, (Münich), september 2001.

[48] L. Seitz and J. Pierson and L. Brunie, "Semantic access control for medical applications in grid environments," (Klagenfurt, Austria). proceedings of Euro-Par 2003.

[49] National Electrical Manufacturers Association, *Digital Imaging and Communications in Medecine (DICOM).* Rosslyn, Virginia, 2001. DICOM 3.

[50] IEEE Storage System Standards Working Group, *Reference Model for Open Storage Systems Interconnection*, september 1994. version 5.

[51] J. Montagnat and F. Bellet and H. Benoit and V. Breton and L. Brunie and H. Duque and Y. Legre and I.E. Magnin et L. Maigne and S. Miguet and

J-M. Pierson and L. Seitz and T. Tweed, "Medical images simulation, storage and processing on the european datagrid testbed," *Journal of Grid Computing (JGC)*, vol. 2, pp. 387–400, december 2004.

[52] A. Oram, ed., *Peer-to-peer : Harnessing the Power of Disruptive Technologies*. Sebastopol, California : O'Reilly, 2001.

[53] B. Cooper and H. Garcia-Molina, "Bidding for storage space in a peer-to-peer data preservation system," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*, (Vienna, Austria), ICDSC, July 2-5 2002.

[54] G. Fedak and C. Germain and V. Ne'ri and F. Cappello, "Xtremweb : A generic global computing system," in *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, (Brisbane, Australia), May 15-18 2001.

[55] R. Buyya and H. Stockinger and J. Giddy and D. Abramson, "Economic models for management of resources in peer-to-peer and grid computing," in *In Proceedings of International Conference on Commercial Applications for High-Performance Computing*, (Denver, Colorado, USA), August 20-24 2001.

[56] R. Buyya and D. Abramson and J. Giddy and H. Stockinger, "Economic models for resource managementand scheduling in grid computing," *The Journal of Concurrency and Computation : Practice and Experience (CCPE)*, May 2002.

[57] L. Gong, "Project jxta : A technology overview," tech. rep., Sun Microsystems Inc, April 2001. http ://www.jxta.org/project/www/docs/TechOverview.pdf.

[58] I. Foster and S. Tuecke and J. Unger, "OGSA data services." Also available as `www.cs.man.ac.uk/grid-db/papers/draft-ggf-dais-dataservices-ggf9.pdf`.

[59] K. Czajkowski and D. Ferguson and I. Foster and J. Frey and S. Graham and T. Maguire and D. Snelling and Steve Tuecke, "From open grid services infrastructure to ws-resource framework : Refactoring & evolution," tech. rep., Fujitsu, IBM and University of Chicago, decembre 2004.

[60] T.A. Howes and M. Smith., "A scalable, deployable directory service framework for the internet," tech. rep., Center for Information Technology Integration, University of Michigan, 1995.

[61] J. Basney and M. Livny, "Deploying a high throughput computing cluster," in *High Performance Cluster Computing*, vol. 1, ch. 5, Prentice Hall PTR, May 1999.

[62] S. Chapinand J. Karpovich and A. Grimshaw, "The legion resource management system," in *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing*, (San Juan, Puerto Rico), April 16 1999.

[63] A. S. Grimshaw, W. A. Wulf, and the whole Legion team, "The legion vision of a worldwide virtual computer," *Communication of the ACM*, vol. 40, pp. 39–45, Jan. 1997.

[64] M. Romberg, "The unicore architecture : seamless access to distributed resources," in *Proceedings of The Eighth International Symposium on High Performance Distributed Computing*, (Redondo Beach, CA, USA), 1999.

[65] J. Almond and D. Snelling, "UNICORE : uniform access to supercomputing as an element of electronic commerce," *Future Generation Computer Systems*, vol. 15, pp. 539–548, Oct. 1999.

[66] R. Buyya and S. Venugopal, "The gridbus toolkit for service oriented grid and utility computing : An overview and status report," in *Proceedings of the First IEEE International Workshop on Grid Economics and Business Models (GECON 2004)*, (Seoul, Korea), pp. 19–36pp, April 23 2004.

[67] Rajasekar and A. and M. Wan, R. Moore and W. Schroeder, "Data grid federation," in *PDPTA, Special Session on New Trends in Distributed Data Access,*, (Las Vegas NV), June 2004.

[68] Arcot Rajasekar et al, "Storage resource broker, managing distributed data in a grid," *Computer Society of India Journal, Special Issue on SAN*, vol. 33, pp. 42–54, October 2003.

[69] D.B. Skillicorn, "The case for datacentric grids," tech. rep., Queen's University, Kingston, Canada, November 2001. Also available as `http://www.cs.queensu.ca/home/skill/papers.html`.

[70] D.B. Skillicorn, "Distributed data-intensive computation and the datacentric grid." `http://www.cs.queensu.ca/home/skill/papers.html`, 2003.

[71] R. Byrom and B. Coghlan and A. Cooke and R. Cordenonsi and L. Cornwall and A. Datta and A. Djaoui and L. Field and S. Fisher and S. Hicks and S. Kenny and J. Magowan and W. Nutt and D. O'Callaghan and M. Oevers and N. Podhorszki and J. Ryan and M. Soni and P. Taylor and A. Wilson and X. Zhu, "The canonical producer : an instrument monitoring component of the relational grid monitoring architecture (r-gma)," in *The 3rd International Symposium on Parallel and Distributed Computing in association with HeteroPar'04*, (University College Cork, Ireland), July 5-7 2004.

[72] A. Oram, ed., *Peer-to-peer : Harnessing the Power of Disruptive Technologies*. Sebastopol, California : O'Reilly, 2001.

[73] C. H. Ding, S. Nutanong, and R. Buyya, "P2P networks for content sharing," tech. rep., Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, December 2003, Feb. 10 2004.

[74] N. Minar, "Distributed systems topologies : Part 1." `http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html`, 2001.

[75] B. Peter and W. Tim and D. Bart and D. Piet, "A comparison of peer-to-peer architectures," tech. rep., Ghent University, Belgium, 2002. Broadband Communication Networks Group (IBCN).

[76] Josh Cates, "Robust and efficient data management for a distributed hash table.," Master's thesis, Massachusetts Institut of Technology (MIT), Boston, USA, 2003.

[77] Russ Cox and Athicha Muthitacharoen and Robert T. Morris, "Serving dns using a peer-to-peer lookup service," in *proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, (Cambridge, MA), March 2002.

[78] D. Barkai, "An introduction to peer-to-peer computing." Also available as `http://www.intel.com/update/departments/initech/it02012.pdf`.

[79] Source Forge, "What is the gift project ?." `http://cvs.sourceforge.net/viewcvs.py/gift/giFT/README?rev=1.9`, Sep 14 2002.

[80] M. Bergner, "Improving performance of modern peer-to-peer services," Master's thesis, UMEA University, Department of Computer Science, Sweden, 2003.

[81] Danny Teaff and Dick Watson and bob Coyne, "The architecture of the high performance storage system," tech. rep., IBM, 1995.

[82] Jon Bakken and Eileen Berman and Chih-Hao Huang and Alexander Moibenko and Don Petravick and Ron Rechenmacher and Kurt Ruthmansdorfer, "Enstore technical design document," tech. rep., Fermilab, 1999. Also available as `http://www-isd.fnal.gov/enstore/design.html`.

[83] D. Walsh and B. Lyon and G. Sager and J. Chang and D. Goldberg and S. Kleiman and T. Lyon and R. Sandberg and and P. Weiss, "Overview of the sun network file system," in *Proceedings of the 1985 Winter Usenix Technical Conference*, January 1985.

[84] P. J. Leach, "A common internet file system (cifs/1.0) protocol," tech. rep., Network Working Group, Internet Engineering Task Force, 1997.

[85] Dave Hitz and James Lau and Michael Malcom, "File system design for an nfs file server appliance," in *USENIX San Francisco 1994 Winter Conference*, (San Francisco), January 1994.

[86] J. H. Morris and M. Satyanarayanan and M. H. Conner and J. H. Howard and D. S. H. Rosenthal and F. D. Smith., "Andrew : A distributed personal computing environment," in *Communications of the ACM*, March 1986.

[87] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S.-T. Tu, and E. R. Zayas, "DEcorum file system architectural overview," in *Proceedings of the Summer 1990 USENIX Conference : June 11–15, 1990, Anaheim, California, USA* (USENIX, ed.), (Berkeley, CA, USA), pp. 151–164, USENIX, Summer 1990.

[88] *Interprocess Communications in Unix; the nooks & crannies.* New Jersey, USA : Prentice Hall PTR, 1998.

[89] *The Design of the UNIX Operating System.* New Jersey, USA : Prentice Hall PTR, 1986.

[90] L. Seitz and J. Pierson and L. Brunie, "Key management for encrypted data storage in distributed systems," in *Proceedings of the second Security In Storage Workshop (SISW)*, (Washington), 2003.

[91] L. Seitz and J. Pierson and L. Brunie, "Semantic access control for medical applications in grid environments," in *Euro-Par 2003 Parallel Processing*, vol. LNCS 2790, pp. 374–383, 2003.

[92] L. Brunie and J-M. Pierson and D. Coquil, "Semantic collaborative web caching," in *Proceedings of the third international conference on Web Information Systems Engineering (WISE'2002)*, (Singapore), pp. 30–39, Dec. 2002.

[93] L. Brunie and D. Coquil, "Enhancement of web proxies using semantic information and cooperation," in *Proceedings of the sixth International Symposium on Programming and Systems (ISPS'2003)*, (Alger, Algeria), Mar. 2003.

[94] N. Pauna, *Evaluation des méthodes de mise en correspondance en imagerie multimodale IRM/TEP thoracique et cardiaque*. PhD thesis, Université Claude Bernanrd Lyon1, 2004.

[95] Quoc Cuong Pham, *Segmentation et mise en correspondance en imagerie cardiaque multimodale conduites par un modele anatomique bi-cavites du cour*. PhD thesis, INSA Lyon, 2002.

[96] C. Lutz, "Reasoning about entity relationship diagrams with complex attribute dependencies," in *Proceedings of the 2002 International Workshop on Description Logics*, 2002.

[97] T. Riccardi, "Data modeling with entity-relationship diagrams." Also available as `http://www.aw.com/info/riccardi/database/Riccardi\_ch4.PDF`.

[98] I. Blanquer and V. Hernandez and F. Mas and D. Segrelles, "A middleware grid for storing, retrieving and processing dicom medical images." DIDAMIC (Distributed Databases and processing in Medical Image Computing) workshop MICCAI, Rennes, Saint-Malo, France, septembre 26-30 2004.

[99] Press, W.H. and Teukolsky, S.A. and Vetterling, W.T. and Flannery, B.P., *Numerical Recipies in C (2nd ed.)*. Cambridge : Cambridge University Press, 1992.

[100] D. Box, Ch. Kindel, B. Grad, *Essential COM*. New York : Addison Wesley, 1998.

[101] William H. Press and William T. Vetterling and Saul A. Teukolsky and Brian P. Flannery, *Numerical Recipes in C++ : the art of scientific computing*. Cambridge : Cambridge University Press, pp. 100-115, 2002.

[102] Ossama Othman, Jaiganesh Balasubramanian, and Douglas C. Schmidt, "Performance evaluation of an adaptive middleware load balancing and monitoring service," in *24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, (Tokyo, Japan), May 23-26 2004.

[103] Jean-Paul Arcangeli and Abdelkader Hameurlain and Frédéric Migeon and Franck Morvan, "Mobile agent based self-adaptive join for wide-area distributed query processing.," *J. Database Manag.*, vol. 15, no. 4, pp. 25–44, 2004.

[104] Pradeep Gore, Douglas C. Schmidt, Carlos O'Ryan, and Ron Cytron, "Designing and optimizing a scalable corba notification service," in *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)*, (Snowbird, Utah), June 18 2001.

[105] Steve Vinoski, "CORBA : Integrating Diverse Applications Within Distributed Heterogeneous Environments," in *IEEE Communications Magazine*, vol. 35, pp. 46–55, February 1997.

[106] Serge Miguet and Jean-Marc Nicod and Jean-Marc Pierson, "A parallel environment for 3d image processing," in *Symposium on Parallel Computing for Solving Large Scale and Irregular Applications (STRATAGEME'96)*, (Sophia-Antipolis, France), pp. 189–199, INRIA, 1996.

[107] K. Hassan, T. Tweed and S. Miguet, "A multi-resolution approach for a content-based image retrieval on the grid. Application to breast cancer detection," in *Methods of Information in Medicine*, vol. 1, pp. 25–33, 2005.

[108] H. Atoui, D. Sarrut and S. Miguet, "Usefulness of image morphing techniques in cancer treatment by conformal radiotherapy," in *SPIE Medical Imaging*, vol. 2, pp. 45–53, 2004.

[109] J-M Pierson, L. Seitz, H. Duque, J. Montagnat, "Meta data for efficient, secure and extensible access to data in a medical grid," in *Database and expert systems applications*, (Zaragoza, Spain), 30 August - 3 september 2004.

[110] S. Atnafu, R. Chbeir, D. Coquil, L. Brunie, "Integrating similarity-based queries in image DBMSs," in *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC)*, (Nicosia, Cyprus), ACM, March 14-17 2004.

[111] I. E. Magnin, J. Montagnat, P. Clarysse, J. Nenonen, and T. Katila, eds., *Functional Imaging and Modeling of the Heart, Second International Workshop, FIMH 2003, Lyon, France, June 5-6, 2003 Proceedings*, vol. 2674 of *Lecture Notes in Computer Science*, 2003.

[112] V. Breton, A.E.Solomonides, R.H.McClatchey, "A perspective on the healthgrid initiative," in *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, (Chicago USA), April 2004.

[113] Benoît Planquelle and Jean-François Méhaut and Nathalie Revol, "Mc-pm^2 : Multi-cluster approach with pm^2.," in *PDPTA*, pp. 779–785, 1999.

[114] Olivier Aumage and Luc Bougé and Jean-François Méhaut and Raymond Namyst, "Madeleine ii : a portable and efficient communication library for high-performance cluster computing.," *Parallel Computing*, vol. 28, no. 4, pp. 607–626, 2002.

[115] Olivier Aumage and Luc Bougé and Lionel Eyraud and Guillaume Mercier and Raymond Namyst and Loïc Prylli and Alexandre Denis and Jean-François Méhaut, "High performance computing on heterogeneous clusters with the madeleine ii communication library.," *Cluster Computing*, vol. 5, no. 1, pp. 43–54, 2002.

[116] V. Breton, P. Clarysse, Y. Germain, E. Jannot, Y. Legre, C. Loomis, J. Montagnat, J-M Moureaux, A. Osorio, X. Pennec and R. Texier, "Grid-enabling medical images analysis," in *Journal of Clinical Monitoring and Computing, Special Issue, to be published*, 2005.

[117] Makela, T. and Pham, Quoc Cuong and Clarysse, P. and Nenonen, J. and Lotjonen, J. and Sipila, O. and Hanninen, H. and Lauerma, K. and Knutti, J. and Katila, T. and Magnin, I.E., "A 3d model-based registration approach for the pet, mr and mcg cardiac data fusion," *Medical Image Analysis*, vol. 7, no. 3, pp. 377–389, 2003.

[118] Pham, Q-C. and Vincent, F. and Clarysse, P. and Croisille, P. and Magnin, I.E., "A fem-based deformable model for the 3d segmentation and tracking of the heart in cardiac mri," in *2nd International Symposium on Image and Signal Processing and Analysis (ISPA 2001)*, (Pula, Croatia), pp. 250–254, 2001.

[119] Xu, C and Prince , J L, "Snakes, shapes, and gradient vector flow," *IEEE Transactions on Image Processing*, vol. 7, no. 3, pp. 359–369, 1998.

[120] Zienkiewicz, O.C. and Taylor, R. L., *La methode des elements finis.* Paris : AFNOR Technique, 1991.

[121] Brecht Claerhout, "From grif to healthgrid : confidentiality and ethical issues," in *HealthGrid*, (Clermont-Ferrand), january 29th - 30th 2004.

[122] Y. Cardenas, JM. Pierson, L. Brunie, "Uniform distributed cache service for grid computing," in *2nd International Workshop on Grid and PeertoPeer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems*, (Copenhagen, Denmark), DEXA2005, August 2005. to be published.

[123] R. Saadi, JM. Pierson, L. Brunie, "Apc : Access pass certificate, distrust certification model for large access in pervasive environement," in *IEEE International Conference on Pervasive Services*, (Santorini, Greece), ICPS'05, July 2005. to be published.

[124] D Cheung-Foo-Wo, J-Y Tigli, and M. Riveill, "Architecture orientée composant et interactions implicites, application aux ordinateurs corporels," in *Cepadues* (Premières Journées Francophones : Mobilité et Ubiquité, ed.), (Sophia Antipolis, France), 1-3 juin 2004.

[125] M. Blay-Fornarino, D. Emsellem, A-M. Pinna-Dery, and M. Riveill, "Un service d'interactions : principes et implémentation," in *RSTI - série TSI*, vol. 2, pp. 175–204, 2004.

11.2 Electronic Links

[126] DataGrid project of the FP5, "http ://eu-datagrid.web.cern.ch/eu-datagrid." The Grid for the European Union, jan. 2001-feb. 2004.

[127] Condor project, "http ://www.cs.wisc.edu/condor." Cycle Stealing Technology for High Throughput Computing, University of Wisconsin at Madison, USA, Professor Miron Livny.

[128] Legion project, "http ://www.cs.virginia.edu/ legion." University of Virginia, USA.

[129] Globus project, "http ://www.globus.org."

[130] Cross Grid project, "http ://www.crossgrid.org." IST Programme of the European Commission, March 1, 2002.

[131] Medi Grid project, "http ://www.creatis.insa-lyon.fr/medigrid."

[132] Castor project, "http ://wwwinfo.cern.ch/pdp/castor." Cern Advance STOrage manager.

[133] HPSS, "http ://www4.clearlake.ibm.com/hpss/index.jsp." High Performance Storage System.

[134] Corba project, "http ://www.omg.org."

[135] CTN, "http ://www.erl.wustl.edu/dicom/ctn.html."

[136] DCMTK Dicom Toolkit, "http ://www.offis.uni-oldenburg.de/projekte/dicom/soft-docs/soft01_e.html."

[137] Globus-OGSA, "http ://www.globus.org/ogsa."

[138] SPITFIRE, "http ://edg-wp2.web.cern.ch/edg-wp2/spitfire/index.html."

[139] MPI Forum, "http ://www.mpi-forum.org/."

[140] Chord project, "http ://www.pdos.lcs.mit.edu/chord/." Flexible lookup primitive for peer-to-peer environments, Frans Kaashoek, Massachussets Institut of Technology (MIT).

[141] PVM project, "http ://www.csm.ornl.gov/pvm/pvm_home.html." Parallel Virtual Machine.

[142] DICOM, "http ://medical.nema.org/." Digital Imaging and COmmunications in Medicine.

[143] European IST project of the FP6, "http ://www.eu-egee.org/." Enabling Grids for E-science and industry in Europe, apr. 2004-mar. 2006.

[144] eDiaMoND project , "http ://www.ediamond.ox.ac.uk/." Oxford University's grid computing project, 1st Dec 2002, 30th November 2004.

[145] e-Science project, "http ://e-science.ox.ac.uk/." Oxford e-Science Centre.

[146] Mammogrid project, "http ://mammogrid.vitamib.com/." 1st 2001.

[147] SMF, "http ://www.mirada-solutions.com/PH185d5.html ?PAGE_ID=741." Standard Mammogram Form.

[148] CREATIS, "http ://www.creatis.insa-lyon.fr/." Centre de Recherche et d'Applications en Traitement de l'Image et du Signal.

[149] LIRIS, "http ://liris.cnrs.fr/." Laboratoire d'Ingénierie des Systémes d'Information.

[150] DISMEDI project, "http ://www.medicaltech.org/dismedi/."

[151] Polytechnic University of Valencia - Spain, "http ://www.grycap.upv.es." High Performance Networking and Computing Group.

[152] CAMAEC project, "http ://www.grycap.upv.es/camaec/." High Performance Networking and Computing Group at Polytechnic University of Valencia - Spain.

[153] EUTIST-M Initiative, "http ://www.medicaltech.org/." IST Programme of the European Commission, IST-1999-20226.

[154] AQUATICS project, "http ://aquatics.crs4.it/public/." IST Programme of the European Commission, IST-1999-20226.

[155] University Hospitals of Geneva, "http ://www.dim.hcuge.ch/03_projects_en.htm." Division of Medical Informatics.

[156] IRMA project, "http ://libra.imib.rwthaachen.de/irma/index_en.php." Image Retrieval in Medical Applications, Germany.

[157] ASSERT project, "http ://rvl2.ecn.purdue.edu/ cbir-dev/www/cbirmain.html."

[158] GRIDS Laboratory, "http ://www.gridbus.org/." Grid Computing and Distributed Systems Laboratory.

[159] Grid Computing Info Center, "http ://www.gridcomputing.com/." Rajkumar Buyya.

[160] LDAP, "http ://www.openldap.org/." Lightweight Directory Access Protocol.

[161] JXTA project, "http ://www.jxta.org/." SUN microsystems, Bill Joy and Mike Clary.

[162] The Global Grid Forum , "http ://www.ggf.org." GGF.

[163] WSRF Project, "http ://www.globus.org/wsrf/."

[164] Globe project, "http ://www.cs.vu.nl/ steen/globe/."

[165] UNICORE project, "http ://unicore.sourceforge.net/." UNICORE (Uniform Interface to Computing Resources).

[166] SRB project, "http ://www.npaci.edu/dice/srb/." Storage Resource Broker, San Diego Supercomputer Center (SDSC).

[167] MCAT project, "http ://www.npaci.edu/dice/software/srb/mcat.html." Metadata Catalog, San Diego Supercomputer Center (SDSC).

[168] LHC Project, "http ://lhc-new-homepage.web.cern.ch/lhc-new-homepage/." Large Hadron Collider (LHC).

[169] LCG Project, "http ://lcg.web.cern.ch/lcg/." LHC Computing Grid.

[170] ARDA Project, "http ://lcg.web.cern.ch/lcg/peb/arda/." Grid Analysis Prototypes of the LHC Experiments, CERN, Switzerland.

[171] K-GRID project, "http ://gridcenter.or.kr/." Korea, 2002 to 2006,.

[172] EUROGRID Project, "http ://www.eurogrid.org/." Nov 1, 2000 - Jan 31, 2004.

[173] The GridCafe, "http ://gridcafe.web.cern.ch/gridcafe/index.html." CERN, Switzerland.

[174] Datacentric Grid Project, "http ://www.cs.queensu.ca/home/skill/datacentric.html." Pr. David Skillicorn, Queen's University, Kingston, Ontario, Canada.

[175] IPG Project, "http ://www.ipg.nasa.gov/." NASA's Information Power Grid (IPG).

[176] Freenet Project, "http ://freenet.sourceforge.net/." Ian Clarke.

[177] Napster Project, "http ://www.napster.com/."

[178] Gnutella Project, "http ://www.gnutella.com/."

[179] Mojeo Nation Project, "http ://sourceforge.net/projects/mojonation."

[180] OpenFT Project, "http ://www.infoanarchy.org/wiki/wiki.pl ?openft."

[181] Oceanstore Project, "http ://oceanstore.cs.berkeley.edu/." UC Berkeley Computer Science Division.

[182] FastTrack Project, "http ://www.slyck.com/ft.php."

[183] Kazaa, "http ://www.kazaa.com/us/index.htm."

[184] Eurostore Project, "http ://eurostore.web.cern.ch/eurostore/." CERN.

[185] Enstore Project, "http ://www-stken.fnal.gov/enstore/." Fermilab, USA.

[186] Disk Cache Project, "http ://www-dcache.desy.de/." DESY.

[187] our team page at Liris, "http ://liris.cnrs.fr/ jpierson/." LIRIS.

[188] GNUPLOT, "http ://www.gnuplot.info/."

[189] Extensible Markup Language XML, "http ://www.xml.org/." Extensible Markup Language XML.

[190] Webster's Dictionary, "http ://www.bennetyee.org/http_webster.cgi."

[191] Object Management Group (OMG), "http ://www.omg.org."

[192] COM : Component Object Model Technologies, "http ://www.microsoft.com/com/default.mspx."

[193] Java Remote Method Invocation (Java RMI), "http ://java.sun.com/products/jdk/rmi/."

[194] Deutsche Elektronen-Synchrotron DESY , "http ://www.desy.de/html/home/."

[195] Grille pour le Traitement d'Informations Médicales (Ragtime), "http ://liris.univ-lyon2.fr/ miguet/ragtime/."

[196] MYSQL, "http ://www.mysql.com/."

[197] HL7 : Health Level 7, "http ://www.hl7.ca/."

[198] medGIFT Project, "http ://www.sim.hcuge.ch/medgift/w01_presentation_fr.htm."

[199] CasImage Project, "http ://www.casimage.com/."

[200] PTM3D Project, "http ://www.limsi.fr/rs99ff/chm99ff/imm99ff/imm4/."

[201] CODA Project, "http ://www.coda.cs.cmu.edu/." Carnegie Mellon University, USA.

[202] Berkeley Open Infrastructure for Network Computing (BOINC), "http ://boinc.berkeley.edu/."