Implementation of Distributed Data Processing in a Database Programming Language

Zongyan Wang

School of Computer Science
McGill University, Montréal
November 2002

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Contents

List of Figures

List of Tables

Résumé

Cette thèse discute la conception et l'exécution des caractères d'intégration d'Internet dans un langage de programmation de base de données JRelix, de sorte qu'elle possède non seulement l'organisation de données, le stockage et les fonctions d'indexation d'un DBMS normal, mais également les possibilités de traitement de données á distance par Internet.

Une prolongation de nom basé sur URL pour les éléments de base de données est adoptée en ce langage de programmation de base de données, qui apporte des possibilités de collaboration et distributives par Internet, sans des changements de syntaxe ou de sémantique indépendamment de la nouvelle structure pour les noms. Les relations, les calculs, les rapports (ou les questions) et l'expression rélationnelle sont traités uniformément comme les éléments de base de données dans notre exécution.

Ces éléments de base de données peuvent être consultés ou exécutés á distance, qui signifie que l'accès aux données ou le traitement de données á distance et le Remote Procedure Call (RPC) sont soutenus. Le partage de ressource est l'accomplissement principal de l'exécution. En outre, l'autonomie de site et le transparent de performance sont accomplis; la gestion basée sur la vue repartie est réalisable; les sites n'ont pas besoin d'être géographiquement éloignés; la gestion de sécurité est mise en application.

Abstract

This thesis discusses the design and implementation of integrating the Internet capability into a database programming language JRelix, so that it not only possesses data organization, storage and indexing capabilities of normal DBMS, but also possesses remote data processing capabilities across the Internet.

A URL-based name extension to database elements in a database programming language is adopted, which gives it collaborative and distributed capability over the Internet with no changes in syntax or semantics apart from the new structure in names. Relations, computations, statements (or queries) and relational expression are treated uniformly as database elements in our implementation. These database elements are enabled to be accessed or executed remotely. As a result, remote data accessing or processing, as well as Remote Procedure Call (RPC) are supported.

Sharing resource is a main achievement of the implementation. In addition, site autonomy and performance transparency are accomplished; distributed view management is provided; sites need not be geographically distant; security management is implemented.

Acknowledgments

First and foremost, I wish to thank my thesis supervisor Professor Tim Merrett for his attentive guidance, invaluable advice, endless patience and continuous encouragement throughout the research and preparation of this thesis. He always provides insights into the implementation and this thesis benefited from his careful reading and constructive criticism. I would also like to thank him for his generous financial support.

I would like to thank Yi Zheng, who provided great information about existing JRelix system and lab operations. I benefited a lot from discussing with her. I also wish to thank the School of Computer Science for the graduate courses and the research environment. Thanks to all the secetaries and system staff for their administrative help and technical assistance.

Special thanks goes to David Noce, who translated the abstract to French and proofread this thesis .

Thanks must also go to my dear parents, mother-in-law and my sister for their endless love and constant support, without which it would be impossible for me to complete my study at McGill .

Finally, I wish to appreciate my husband, Qiang Xu, for his understanding, love, support and encouragement during my study.

Chapter 1

Introduction

This thesis describes the design and implementation of a distributed system with collaborative and distribution capability in a database programming language JRelix [Yua98, Bak98, Hao98, He97, Sun00].

In this chapter, we will give the background and preliminary material needed throughout the thesis, as well as a brief introduction to the approach used for the implementation. In section 1.1, we will present the purpose of designing and implementing this distributed system in JRelix. In section 1.2, some early DBMS prototypes as well as main principles of DBMS are illustrated. However, one thing needs to be noted; our implementation is not going to replace distributed database systems or build a new dustributed database system. Instead, our implementaion is to build a low-level mechanism for dustributed data processing in a database programming language, upon which we can develope applications with distributed database systems functions. Since in our implementation, URL-based name structure is adopted and our approach resembles HTTP in some aspects, a brief introduction to HTTP and URL is presented in section 1.3. Section 1.4 is an overview of JRelix system on which our work is based. In section 1.5, we will describe the approach used in our implementation to integrate the Internet capability into a database programming language. In section 1.6, we will give a brief outline of the thesis.

1.1 Purpose of the Thesis

Networks of computers are everywhere. The Internet is a vast interconnected collection of computer networks of many different types, in which components located at networked computers communicate and coordinate their actions. All the databases that are linked by the Internet may be regarded as an enormous database family. They cooperate with each other, share public information, and private data is protected from malicious intruders.

Distributed database technology is one of the most important developments of the past decades. The maturation of database management systems (DBMS) technology has coincided with significant developments in distributed systems and the result is the emergence of distributed DBMSs. The sharing of resource is a main motivation for constructing distributed database systems. Other basic motivations for distributing databases are improved performance and increased availability.

The purpose of this thesis is to integrate the Internet capability into a high level database programming language JRelix, developed at the Aldat lab of School of Computer Science at McGill University, so that it not only possesses the data organization, storage and indexing capabilities of the normal DBMS, but also possesses the remote data processing capabilities of the Internet.

1.2 Distributed Database

1.2.1 Early Distributed DBMS Prototypes

Several distributed DBMS prototypes were developed during the 1970s and early 1980s, such as SDD-1 [Rot80, RG77], `R*` [Wil81], Distributed INGRES [Sto86a] , DDM [Cha83], POREL [NW82], SIRIUS-DELTA [Lit82], MULTIBASE [Smi82] and DDTS [DW80]. All extended single site DBMSs to manage relations that were spread over the sites in a computer network .

SDD-1

The SDD-1 project [Rot80, RG77], developed at the Computer Corporation of America, was the first prototype of a distributed database system; it was designed between 1976 and 1978, and implemented in 1979. This project made a significant contribution to the DBMS research field. The SDD-1 project is a pioneering progject which helps understanding of the important problems of distributed database.

SDD-1 supports the relational data model. Global relations can be fragmented in two steps, first horizontally and then vertically; fragments can be replicated. SDD-1 provides fragmentation transparency, i.e., the user is unaware of fragments and their location. Concurrency control in SDD-1 [BRGP78, BSR80, BS80] uses the conservative timestamp method enhanced by several additional characteristics. In SDD-1 query processing [GBW+81], semijoins are used for reducing cardinalities of relations. Reliability in SDD-1 [HS80] is provided by a virtual machine which has a layered software architecture. Many of the ideas such as fragmentation, semi-join, timestamps for currency control were proposed and used for the first time in the SDD-1 project.

However the performance of the SDD-1 is not ideal. In particular, concurrency control is not deadlock-free; the data manipulation language used in SDD-1 introduces some limitations in query processing; the reliability system does not survive network partitions.

IBM system R*

System R is an experimental database management system designed and built by members of the IBM San Jose Research Laboratory (now IBM Almaden Research Center) in the 1970's. It is a research program on the relational model of data. `R*` [Wil81] is the distributed version of the original System R prototype. The goal of the R* project is to build a distributed database system. Each site is an autonomous relational database system cooperating with other sites.

Data in R* is stored in relations. In R*, sites need not be geographically distant:

different sites can be on the same computer. This is considerablly important not only for the development and testing of the database applications, but also for application systems. Different R* modules can be placed on the same computer, which is helpful for security, accounting, or performance reasons.

One of the most important objectives of R* is to provide site autonomy [Lin80]. Each site is able to control other sites' accessing to its own data as well as to manipulate its data without being conditioned by any other site. The first goal is completely achieved by R*. However, the second goal is only partially achieved. Since R* uses two-phase-commitment of transactions [Lin83], a loss of site autonomy cannot be avoided. Site autonomy also requires that the system be able to grow incrementally and to operate continuously. New sites can join to existing ones without requir existing sites to agree with joining sites on global data structures or definitions.

Another important issue in R* is location transparency (the user is not aware of the actual location of data). Thus, from the programmers' viewpoint, the use of R* is basically equivalent to the use of centralized system. An R* *system wide* name [Lin81] structure is as follows

```
<creator>@<creator-site>.<object>@<birth-site>
```

Synonyms and defaults are used for simplifying this naming scheme.

In R*, view management is also distributed [BHL83]. Views can be defined using relations which residing at a different site from the definition site. Fragmentation and replication are not implemented by the R* system. In fact, Views can simulate fragmentation. A global relation can be defined as a view built on top of several relations at different sites which represent fragments.

As it will be shown, our approach supports most of the features described above. Further more, we incorporate more technologies, e.g. remote procedure call or remote method invocation, and distributed event-based system.

Distributed INGRES

Distributed INGRES [Sto86a, Sto86b] was developed at the University of California at Berkeley. It is a distributed version of the relational database system INGRES. Distributed INGRES is designed to operate on both local (Ethernet-like) network and geographical network. Some aspects of query processing are parametric with respect to the type of network.

Distributed INGRES provides fragmentation and location transparency. Horizontal fragmentation is supported, while vertical fragmentation is not. Fragments can be replicated; one of them is designated as primary, which is used by transaction management, concurrency control and reliability algorithms.

Distributed INGRES uses 2-phase-locking for concurrency control. Deadlocks are detected and resolved with a centralized deadlock detector. Global query processing [ESW78b] in Distributed INGRES extends the decomposition strategy used for single-site INGRES.

OTHERS

Other homogeneous distributed database systems prototypes are developed in 1970s and 1980s: DDM [Cha83], developed at the Computer Corporation of America, POREL [NW82], developed at the University of Stuttgart, and SIRIUS-DELTA [Lit82], developed at INRIA.

Two major research prototypes in the field of heterogeneous distributed database system are MULTIBASE [Smi82], developed at Computer Corporation of America, and DDTS [DW80], developed at Honeywll Corporate Computer Science Center. The most ambitious requirement of heterogeneous systems is the capability of providing DBMS independence. i.e. DBMS instances at different sites all support the same interface and could all participate somehow in a distributed system. This is a very difficult goal, thus some prototypes do not attain this overall objective. The MULTIBASE system is developed for providing transparency to retrieval applications, while updates are performed by each individual DBMS, without coordination.

1.2.2 Principles of Distributed Database Overview

Survey papers discussing major issues concerning distributed systems has been written by Rothnie and Goodman [RJG77], Bernstein et al [BRJS78] and Gray [Gra79]. Textbook discussions are offered by Ceri and Pelagatti [CP84], Ozsu and Valduriez [OV99], Date [Dat00], Silberschatz and Korth and Sudarshan [SKS97]. Our survey is mainly based on these materials and relevant literatures.

Data distribution alternatives

Consider a relation R that is to be stored in the database. There are several approaches to store this relation in the distributed database.

- **Replication**. The system maintains several identical replicas(copies) of the relation. Each replica is stored at a different site, resulting in data replication.
- **Fragmentation**. The relation is partitioned into several fragments. Each fragment is stored at a different site.
- **Replication and fragmentation**. The relation is partitioned into several fragments. The system maintains several replicas of each fragment.

Replication

If relation R is replicated, a copy of R is stored in two or more sites. In the most extreme case, we have full replication, in which a copy is stored in every site in the system. On the other hand, an alternative to replication is to store only one copy of relation R. There are some advantages and disadvantages for replication.

- **Enhanced performance**. Replication helps performance since diverse and conflicting user requirements can be more easy to deal with. For example, data that is frequently accessed by one user can be placed on that users local machine.
- **Availability**. If one of the machines fails, a copy of the data is still available on another machine on the network.

- **Increased overhead on update**. The system must ensure that all replicas of a relation R are consistent; otherwise, errors may result. Thus, whenever R is updated, the update must be propagated to all sites containing replicas.

Fragmentation

If relation R is fragmented, R is divided into a number of fragments R1, R2,...,Rn. There are two different schemes for fragmenting a relation: horizontal fragmentation and vertical fragmentation.

Horizontal fragmentation of a relation is accomplished by a selection operation. The selection operation places each tuple of the relation in a different partition based on a predicate (e.g. an employee relation may be fragmented according to the department of the employees). We can obtain the reconstruction of the relation R by taking unions of all fragments.

```
R = R1 union R2 union ... union Rn
```

Figure 1.1 shows an example of horizontal fragmentation of relation Employee.

```
Employee                                    Employee1
(id   name  department  salary )            (id   name  department  salary)
 1    Joe      CS        50000               1    Joe      CS        50000
 2    Sam      CS        55000               2    Sam      CS        55000
 3    Sue      EE        55000
 4    Joe      EE        45000                Employee2
                                            (id    name  department  salary)
                                             3    Sue      EE        55000
                                             4    Joe      EE        45000
```

Figure 1.1: Horizontal fragmentation of relation: Employee

Vertical fragmentation divides a relation into a number of fragments by projecting over its attributes, such that R can be reconstructed from the fragments by taking natural joins.

```
R = R1 natjoin R2 natjoin ... natjoin Rn
```

Figure 1.2 shows an example of vertical fragmentation of relation Employee.

Employee

(id	name	department	salary)
1	Joe	CS	50000
2	Sam	CS	55000
3	Sue	EE	55000
4	Joe	EE	45000

Employee1

(id	name	salary)
1	Joe	50000
2	Sam	55000
3	Sue	55000
4	Joe	45000

Employee2

(id	department)
1	CS
2	CS
3	EE
4	EE

Figure 1.2: Vertical fragmentation of relation: Employee

Fragmentation is desirable because it places data in a close proximity to its place of use, thus potentially reducing transmission cost. In addition, it reduces the size of relations that are involved in user queries.

Replication and Fragmentation

Based on the user access patterns, each of the fragments may also be replicated. This is preferable when the same data was accessed from applications that run at a number of sites. In this case, it may be more cost-effective to duplicate the data at a number of sites rather than continuously transmitng it between them.

Objectives of DDBS

1. **Site autonomy & No reliance on central site**

Under many situations, site autonomy is desired. Local data is locally owned and managed. All data really belongs to some local database, even if it is accessible from other remote sites. Such matters as security, integrity, and storage of local data are controled by local DBMS. To manipulate local data should not be conditioned by any other sites.

Site autonomy also requires that the system be able to grow incrementally and to operate continuously. That menas new sites can join to existing ones without requiring existing sites to agree with joining sites on global data structures or definitions.

Site autonomy implies that all sites must be treated as equals. There are must not be any reliance on a central *master* site for some central service, e.g. centralized query processing, centralized transaction management, or centralized naming services. Reliance on central site would be undesirable for at least two reasons. First, the central site might be a bottleneck. Second, the system would be vulnerable. If the central site went down, the whole system would be down.

2. **Transparency**

The possible forms of transparency discussed in [SH] and [OV99] can be summarized as follows:

- Location transparency (network transparency or distribution transparency): users do not have to specify where data is located.
- Replication transparency: objects can be copied, and copies are maintained automatically. Users should be able to behave as if the data were in fact not replicated at all.
- Fragment transparency: tables can be fragmented to different sites. Users should be able to behave as if the data were in fact not fragmented at all.
- DBMS transparency: should not matter what DBMS is running at each site
- Performance transparency: performance independent of submission site
- Transaction transparency: looks like single-site transactions

3. **Improved reliability**

Distributed DBMS are intended to improve reliability since they have replicated components, thus eliminating single points-of-failure. The failure of a single site, or the failure of a communication that makes one or more sites unreachable, may not fatal enough to bring down the entire system. Users may be permitted to access other parts of the distributed database.

4. **Enhanced performance**

Data replication and data fragmentation enables data localization. This has two potential advantages:

- Since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for a centralized database, and
- Localization reduces remote access delays and data movement among sites.

The parallel capability of distributed database system may be achieved by inter-query and intra-query parallelism. Inter-query parallelism results from the ability to execute multiple queries at the same time. Intra-query parallelism is achieved by breaking up a single query into a number of subqueries. Each of them is executed at a different site, accessing a different part of the distributed database.

Technical Issues Overview

Distributed query processing

Distributed query processing is discussed in [Won77, ESW78a, HY79, ES80, CP83, Won83]. An important aspect of distributed query processing is query optimization. The problem is how to decide on a strategy for executing each query over the network in the most cost-effective way. The factors to be considered are the distribution of data and communication costs. The objective is to optimize parallelism of the distributed system to enhance the performance of executing the query [OV99]. An important aspect of query optimization is join ordering, since permutations of joins within the query may lead to improvements of several orders of magnitude. One basic technique for optimizing a sequence of distributed join operation is through use of the semijoin operator [BC81, BG81b, KYY82]. The main value of the semijoin is to reduce the size of the join operands and thus the communication cost. However, they might increase local costs.

Distributed transaction management

The user accesses to shared databases are formulated as transactions [BN97, BHG87], which are units of execution that satisfy four properties: atomicity, consistency, isolation, and durability—jointly known as the ACID properties [Gra78, GR93]. Atomicity means transactions are atomic (all or nothing). Consistency means a transaction transforms a consistent state of database into another consistent state. Isolation means transactions are isolated from one another. Durability means once a transaction commits, its update survive in the database, even if there is a subsequent system crash. ACID properties are enforced by means of concurrency control and reliability protocols.

Distributed concurrency control

Papers covering distributed concurrency control are offered by [BS80, BSR80, BG81a, BG82]. Concurrency control involves the synchronization of access the distributed database, such that the integrity of the database is maintained. The synchronization is achieved by concurrency control algorithms. The concurrency control problem in a distributed context is somewhat different from the centralized framework. One not only has to worry about the integrity of a single database, but also about the consistency of multiple copies of the database. Distributed concurrency control algorithms enforce global serializability, that is "the execution of the set of transactions at each site is serializable and the serialization orders of these transactions at all these sites are identical" [OV99]. Most concurrency control algorithms are locking-based, e.g. centralized locking, primary copying locking, and distributed locking algorithm. It is a well-known theorem that if DBMS obey the two-phase locking rule, "no lock on behalf of a transaction should be set once a lock previously held by the transaction is released", then it is possible to ensure the serializability. The side effect is that they cause dead locks. Distributed deadlock-detection algorithms are presented by [RSL78, CHM83]. [Kna87] surveys the distributed deadlock detection literature.

Distributed reliability protocols.

Distributed database system are potentially more reliable since there are multiple copies of each system component, which eliminates single points-of-failure. Data is replicated to ensure that data is accessable in case of system failures. Distributed reliability protocols maintain the atomicity and durability properties. To ensure atomicity, the transaction must execute a commit protocol. The most widely used commit protocol is the two-phase commit protocol (2PC) [LS76, ML83]. 2PC may lead to blocking: a situation in which the fate of a transaction cannot be determined until a failed site (the coordinator) recovers. To avoid blocking, we can use three-phase commit protocol (3PC) [Ske81] and modified version of 2PC [ML83]. To ensure durability, distributed recovery [ABG84, Koh81] ensure that the system can recover to a consistent state following a failure.

We have introudeced some early distributed DBMS prototypes, main features and some technical issues of DBMS in this section. The purpose of this thesis is not to replace distributed DBMS or build a new one. In our implementation, we provide the Internet capability or distributed data processing capability into a database programming language JRelix. Our implementation achieves some features of distributed DBMS and could be used with other language features of JRelix to implement DDBMs (see Chapter 5).

1.2.3 Oracle 9i Distributed Database System

Oracle supports both homogenous and heterogeneous architectures. In a **homogenous distributed database system**, each database is an Oracle database. In a **heterogeneous distributed database system**, at least one of the databases is a non-Oracle database. The Oracle database server accesses the non-Oracle system using Oracle Heterogeneous Services in conjunction with an **agent.** If a user accesses the non-Oracle data store using an Oracle Transparent Gateway, then the agent is a system-specific application. The agent is specific to the non-Oracle system, so each type of system requires a different agent. For example, if a user includes a Sybase database in an Oracle distributed system, then we need to obtain a Sybase-specific transparent gateway so that the Oracle databases in the system can communicate with it. Alternatively, we can use **generic connectivity** to access non-Oracle data stores so long as the non-Oracle system supports the ODBC or OLE DB protocols.

Database Links

Oracle's distributed database management system architecture lets us access data in remote databases using Oracle Net and an Oracle server. We can identify a remote table, view, or materialized view by appending `@`*dblink* to the end of its name. A database link is a connection between two physical database servers that allows a client to access them as one logical database. In Oracle, a database link is a pointer that defines a one-way communication path from an Oracle database server to another database server. The link pointer is actually defined as an entry in a data dictionary table. To access the link, we must be connected to the local database that contains the data dictionary entry.

Oracle lets us create **private, public**, and **global** database links. If they are private, then only the user who created the link has access; if they are public, then all database users have access. Global database links create network-wide links for every Oracle database in the network. When we create a private or public database link, we can determine which schema on the remote database the link will establish connections to by creating fixed user, current user, or connected user database links.

The great advantage of database links is that they allow users to access another user's objects in a remote database so that they are bounded by the privilege set of the object's owner. Database links allow us to grant limited access on remote databases to local users.

Every application that references a remote server using a standard database link establishes a connection between the local database and the remote database. Many users running applications simultaneously can cause a high number of connections between the local and remote databases. Shared database links enable us to limit the number of network connections required between the local server and the remote server.

Location Transparency

Typically, administrators and developers use synonyms to establish location transparency for the tables and supporting objects in an application schema. For example, the following statements create synonyms in a database for tables in another, remote database.

```
CREATE PUBLIC SYNONYM emp
  FOR scott.emp@sales.us.americas.acme_auto.com;
CREATE PUBLIC SYNONYM dept
  FOR scott.dept@sales.us.americas.acme_auto.com;
```

Now, an application can issue a query that does not have to account for the location of the remote tables.

```
SELECT ename, dname
  FROM emp e, dept d
  WHERE e.deptno = d.deptno;
```

In addition to synonyms, developers can also use views and stored procedures to establish location transparency for applications that work in a distributed database system.

Local views can provide location transparency for local and remote tables in a distributed database system. For example, assume that table emp is stored in a local database and table dept is stored in a remote database. To make these tables transparent to users of the system, we can create a view in the local database that joins local and remote data:

```
CREATE VIEW company AS
   SELECT a.empno, a.ename, b.dname
   FROM scott.emp a, jward.dept@hq.acme.com b
   WHERE a.deptno = b.deptno;
```

PL/SQL program units called procedures can also provide location transparency. We have three options:

1) Using Local Procedures to Reference Remote Data

Procedures or functions (either standalone or in packages) can contain SQL statements that reference remote data. For example, consider the procedure created by the following statement:

```
CREATE PROCEDURE fire_emp (enum NUMBER) AS
BEGIN
  DELETE FROM emp@hq.acme.com
  WHERE empno = enum;
END;
```

2) Using Local Procedures to Call Remote Procedures

You can use a local procedure to call a remote procedure. The remote procedure can then execute the required DML. For example, assume that scott connects to `local_db` and creates the following procedure:

```
CREATE PROCEDURE fire_emp (enum NUMBER)
AS
BEGIN
  EXECUTE term_emp@hq.acme.com;
END;
```

3) Using Local Synonyms to Reference Remote Procedures

SQL and COMMIT Transparency

Oracle's distributed database architecture also provides query, update, and transaction transparency. For example, standard SQL statements such as SELECT, INSERT, UPDATE, and DELETE work just as they do in a nondistributed database environment. Additionally, applications control transactions using the standard SQL statements COMMIT, SAVEPOINT, and ROLLBACK–there is no requirement for complex programming or other special operations to provide distributed transaction control.

- The statements in a single transaction can reference any number of local or remote tables.
- Oracle guarantees that all nodes involved in a distributed transaction take the same action: they either all commit or all roll back the transaction.

- If a network or system failure occurs during the commit of a distributed transaction, the transaction is automatically and transparently resolved globally. Specifically, when the network or system is restored, the nodes either all commit or all roll back the transaction.

Replication Transparency

Oracle also provide many features to transparently replicate data among the nodes of the system. For more information about Oracle's replication features, see Oracle9i online documents [Ora02b, Ora02a].

Remote Procedure Calls (RPCs)

Developers can code PL/SQL packages and procedures to support applications that work with a distributed database. Applications can make local procedure calls to perform work at the local database and remote procedure calls (RPCs) to perform work at a remote database.

When a program calls a remote procedure, the local server passes all procedure parameters to the remote server in the call. For example, the following PL/SQL program unit calls the packaged procedure del_emp located at the remote sales database and passes it the parameter 1257:

```
BEGIN
 emp_mgmt.del_emp@sales.us.americas.acme_auto.com(1257);
END;
```

In order for the RPC to succeed, the called procedure must exist at the remote site, and the user being connected to must have the proper privileges to execute the procedure. Local users can also connect to the remote database and create remote procedures:

```
CONNECT scott/tiger@hq.acme.com

CREATE PROCEDURE term_emp (enum NUMBER)
AS
BEGIN
  DELETE FROM emp WHERE empno = enum;
END;
```

Distributed Query Optimization

Distributed query optimization is an Oracle feature that reduces the amount of data transfer required between sites.

Distributed query optimization uses Oracle's cost-based optimization to find or generate SQL expressions that extract only the necessary data from remote tables, process that data at a remote site or sometimes at the local site, and send the results to the local site for final processing. This operation reduces the amount of required data transfer when compared to the time it takes to transfer all the table data to the local site for processing.

Site Autonomy

Although several Oracle databases can work together, each database is a separate repository of data that is managed individually. However, users should not ignore the global requirements of the system.

Restrictions on Distributed Queries

Several restrictions apply to Oracle distributed queries.

- Within a single SQL statement, updated tables, and locked tables must be located at the same node, as well as all referenced LONG and LONG RAW columns, sequences.

  For example, the following statement will raise an error:

  ```
  SELECT employees_ny.*
  FROM employees_ny@ny, departments
  WHERE employees_ny.department_id = departments.department_id
  AND departments.department_name = 'ACCOUNTING'
  FOR UPDATE OF employees_ny.salary;
  ```

- Oracle does not allow remote DDL statements (for example, CREATE, ALTER, and DROP) in homogeneous systems. Note that in Heterogeneous Systems, a pass-through facility allows users to execute DDL.

- In a distributed database system, Oracle always evaluates environmentally-dependent SQL functions such as SYSDATE, USER, UID, and USERENV with respect to the local server, no matter where the statement (or portion of a statement) executes.
- A number of performance restrictions relate to access of remote objects:
  - Remote views do not have statistical data.
  - Queries on partitioned tables might not be optimized.
  - No more than 20 indexes are considered for a remote table.
  - No more than 20 columns are used for a composite index.
- There is a restriction in Oracle's implementation of distributed read consistency.

1.3 HTTP

The protocol used in our approach is similar to HTTP. In this section, we present a brief introduction to the paradigm of HTTP and URL. In section 1.5, the difference and similarity between HTTP and our protocol ALDATP is described in detail.

HTTP Overview

HTTP stands for **Hypertext Transfer Protocol**. It is the network protocol used to deliver files and data (collectively called *resources*) on the World Wide Web, whether they are HTML files, image files, query results, or anything else. Usually, HTTP takes place through TCP/IP sockets. HTTP has been in use since 1990. The original version HTTP/0.9 has no RFC. RFC 1945 [BLFF96] gives HTTP/1.0 in 1996 and it is replaced by RFC 2616 [FGM+99] which specifies HTTP/1.1 in 1999.

A simplified organization of the Web is shown in figure 1.3

The Web client (browser) communicates with a Web server using one or more TCP connections. The well-known port for the Web server is TCP port 80, but other ports can be used. The protocol used by the client and server to communicate over

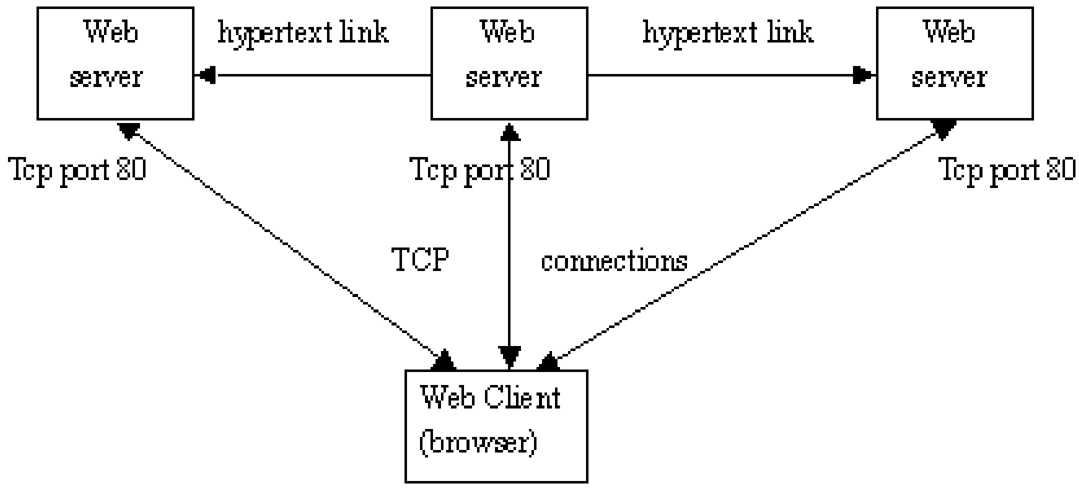


Figure 1.3: Organization of a Web client-server

the TCP connection is called HTTP. The client establishes a TCP connection to the server, issues a request, and reads back the servers response.

Web server can "point to" other Web servers with hypertext links. The file returned by the server normally contains pointers to other files that can reside on other servers. It is the client(browser) that open other TCP connections to follow these links from server to server to fetch the files. These links are not restricted to pointing only to other Web servers. They can point to an FTP server or a Telnet server.

The early versions of HTTP, i.e. HTTP/0.9 and HTTP/1.0, make a new connection for each transfer. The server denotes the end of its response by closing the connection. The persistent connection extension added into HTTP/1.1 was motivated by effenciency concerns. A connection may be used for one or more request/response exchanges.

URL

HTTP is used to transmit resources, not just files. A resource is some chunk of information that can be identified by a URL: a Uniform Resource Locator. The specification and meaning of URLs is given in RFC 1738 [BLMM94] and RFC 1808 [Fie95].

URLs are part of a grander scheme called URIs(Uniform Resource Identifiers). URIs are described in RFC 1630 [BL94]. The most common kind of resource is a file, but a resource may also be a dynamically generated query result, the output of a CGI script, or something else.

Every URL, in its full, absolute form, has two top-level components:

```
scheme : scheme-specific-location
```

The first component, the *scheme*, declares which type of URL this is. For instance, *mailto, ftp, telnet, http* are examples of schemes. The second component specifies resources, e.g files or CGI (or JSP) codes, and the scheme specifies the communication protocol to retrieve them. So we can define our own scheme if we invent a useful new protocol.

HTTP URLs are the most widely used. An HTTP URL has two main jobs to do: to identify which web server maintains the resource, and to identify which of the resources at that server is required. In general,

HTTP URLs are of the following form:

```
http://servername [:port] [/pathNameOnServer] [?arguments]
```

Items in square brackets are optional. A full HTTP URL always begins with the string `"http://"` followed by a server name. The server name is optionally followed by the number of the *port* on which the server listens for requests. Then comes an optional path name of the server's resource. If this is absent then the server's default web page is required. Finally, the URL optionally ends in a set of arguments.

To publish a resource on the Web, a user must first place the corresponding file in a directory that the web server can access. Knowing the name of the server S and a path name for the file P that the server can recognize, the user can then construct the URL as `"http://S/P"`.

There are certain pathname conventions that servers recognize. For example, a pathname beginning `"~tim"` is by convention in a subdirectory `"public_html"` of user Tim's home directory.

HTTP Protocols

Message Types: Requests and responses

There are two HTTP/1.1 message types: requests and responses. The format of an HTTP/1.1 request is

```
Method request-URI HTTP-version
Headers
<blank line>
[message-body]
```

The format of an HTTP/1.1 response is

```
HTTP-version status-code reason-phrase
Header
<blank line>
[message-body]
```

Methods

The following methods are supported by HTTP/1.1.

1. The **GET** method is the most common HTTP method, which returns what ever information is identified by the request-URI.

2. The **HEAD** request is similar to the **GET** request, but only the server's header information is returned, not the actual contents(i.e. no message-body) of the specified document. This request is often used to test a hypertext link for validity, accessibility, and recent modification.

3. A **POST** request is used to send data to the server to be processed in some way, such as by a CGI script. A POST request is different from a GET request in the following ways:

- A block of data is sent with the request, in the message body. There are usually extra headers to describe this message body, suc as **Content-Type:** and **Content-Length:**.

- The request URI is not a resource to retrieve; it is usually a program to handle the data you're sending.

- The HTTP response is normally a program output, not a static file.

Besides "GET", "HEAD", "POST", HTTP/1.1 supports new methods, such as "OPTIONS", "PUT", "DELETE", "TRACE".

Header Fields

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method invocation.

The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

Status Codes

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user.

1.4 JRelix

Relational Database System

The relational model was first proposed by Dr. E.F.Codd in his pioneering paper "A Relational Model of Data for Large Shared Data Banks" [Cod70]. Since then. relational database systems have developed rapidly. In Codd's relational model, a collection of tables that he terms relations are used to model and store data. Each relation resembles a table which consists of rows and columns. "tuples" are used to refer to rows and "attributes" are used to refer to the column headers. The term "domain" refers to the set of legal values that an attributes can have, i.e. the data type of an attribute.

Operations on Relations

Relational Algebra, which is also proposed by Codd, consists of a set of operations applied on relations for retrieving information. In the relational algebra, there is no operation performed on individual tuples. The relational operators take relations as operands and return a relation as a result which can be further manipulated.

The relational algebra operations are usually classified as unary or binary, according to the number of operands. Unary operators take a single relation as operand and binary operators take two relations as operands. Both of them produce a single relation as their result.

Unary operations

Projection: makes a copy of a relation with a specific subset of the attributes

Selection: selects tuples that satisfy a specific condition

Binary operations

mu-join: join operators that generalize set-valued set operations

sigma-join: join operators that generalize logic-valued set operations

Operations on Domains

Merrett[Mer84] proposed the domain algebra which consists of a set of operations that enable the arithmetic and related processing of the values of attributes in individual tuples. It allows the user to create new domains from existing ones. The generation of a new value from many values within a tuple or from values along an attribute also becomes possible. The domain algebra operations are defined as follows:

Horizontal operations: new value is generated from the values with a tuple
Constant
Rename
Function
If-then-else

Vertical Operations: new value is generated from values along an attribute
Reduction
Equivalence Reduction
Functional Mapping
Partial Functional Mapping

Database Programming Language

The relational model has proven itself exceptionally useful for many applications. However, the commercial implementations of the relational model are lacking in expressive power and in the ability to handle complex data. For many applications such as Computer automated design (CAD), VLSI chips design and Geographic Information Systems (GIS), these implementations are inadequate tools. The relational model itself, however, is not limited to these implementations. This has led to research in the field of database programming language (DBPL). DBMS are capable of dealing with large amounts of persistent data, while programming languages provide well-proven and powerful techniques for creating, organizing and manipulating data in memory. DBPLs seek to integrate the technologies and paradigms of programming languages and database management in order to solve the problem of developing data-intensive applications.

One of the approaches to create a database programming language was to embed a database query language into an existing programming language. The INGRES relational database system [SWKH76] embedded its query language QUEL into the C programming language to produce the EQUEL language. This paradigm, e.g. SQL plus JDBC plus Java, is still being widely used by most commercial DBMSs like DB2, Oracle, Informix, SQL Server etc. A major disadvantage of this approach is that it requires the programmer to be fluent with both the host language and the query language. It also yields an awkward programming environment by mixing the types of query language together with typing systems of the host language.

Another approach to creating a DBPL is to add database features to existing programming language. For example, Pascal/R [Sch77] combines the relational data model with the Pascal. One more approach aimed to design a programming language with uniform persistence. In a persistent Programming Language, such as Psalgol [Mor88, ABC$^+$83], data of any type, e.g. arrays and records used in primary memory and abstraction of a relation or file used on the persistent storage, may persist. ObjectStore [LLOW91] adds persistence to the C++ programming language which makes the accessing of persistent data seamless to the programmer.

JRelix (the Java implementation of a Relational database programming language in Unix) was developed at the Aldat lab of the School of Computer Science at McGill University. JRelix contains a database management systems (DBMS) which is responsible for organizing and storing data, and a programming language Aldat(Algebraic Data Language), based on relational algebra and domain algebra [Mer84, Yua98]. JRelix incorporates complex constructs such as computations (functions and procedures), some object-oriented paradigms, such as instantiation [Bak98, Zhe02], and nested relations [He97, Hao98]. The event handler [Sun00], which is a characteristic of active database systems, and attribute metadata [Mer01] for relational OLAP and data mining are implemented in JRelix. A GIS Editor [Che01] is also implemented which demonstrates JRelix capability to support sophisticated data.

1.5 Approach in the thesis

In this section, we are going to present our approach ALDATP for implementing a distributed system. We first illustrate why we implement ALDATP and then outline what is ALDATP.

1.5.1 Why ALDATP

There are several reasons for building distributed database systems, including sharing of data, reliability and availability, and speedup of query processing. However, along with these advantages come several disadvantages, including higher software-development cost, greater potential for bugs, and increased processing overhead. The primary disadvantage of distributed database system is the added complexity required to ensure proper coordination among the sites. Up untill now, nobody provides all the characteristics listed in section 1.2.2. Different implementation will attach different degrees of importance to different objectives in different environments.

There are several paradigms and associated protocols which might offer bases for adding network programming into database programming. The first alternative is TCP data transmission protocol [Ste96, Ste94] . But the user or programmer must be fluent with a set of protocols or messages for various data communications if this paradigm is adopted for adding network programming into database programming. However, this can be avoided and we can make the protocols totally transparent in a high level implementation. Another disadvantage is the need to deal with port numbers makes it awkward when multiple database systems coexist on one physical computer, thus many and even dynamical port numbers must be involved on each host. Perhaps several new syntaxes need to be added; this depends on the existing database programming language.

Remote invocation is another programming models for distributed application.

Such programs need to be able to invoke operations in other process, often running in different computers. The earliest and perhaps the best-known of these was the extension of the conventional procedure call model to the remote procedure call (RPC) model [DJ84, Rao95], which allows client programs to call procedure in server programs running in separate process and generally in different computers from the client. The most well-known RPC package is Sun RPC. RFC 1831 [SM95] describes Sun RPC which was designed for client-server communication in Sun NFS network file system.

More recently, the object-based programming model has been extended to allow objects in different process to communicate with one another by means of remote method invocation (RMI) [CDK01]. Java RMI [WW] is the mostly adopted RMI package. The Object Management Group's Common Object Request Broker Architecture (CORBA) [Gro96, Tib95] is designed to support the role of an object request broker that enables client to invoke methods in remote objects, while both clients and servers can be implemented in a variety of programming languages. The basic idea for RPC, RMI and CORBA are the same, which is remote invocation.

In JRelix, computation can be invoked by means of a top-level call taking domains or relations as its parameters or taking no parameters. Thus computations function as procedures, which is a top level procedural abstraction. In our database programming language, we treat relations, computations, statements(or queries) and relational expression uniformly as database elements. Since we enable these database elements to be accessed or executed remotely, we get remote invocation capability as a result. Furthermore, it is a more capable paradigm since it not only provides RPC and RMI but also remote data accessing.

Merrett [Mer02] suggests adopting a URL-based name extension to database elements in our database programming language "ALDAT", which gives it collaborative and distributed capability over the Internet with no changes in syntax or semantics apart from the new structure in names, thus makes maximum use of minimal ideas. We call this approach "ALDATP" (**aldat p**rotocol).

1.5.2 What is ALDATP

Basically, the URL-based name structure in ALDATP is of the following form:

aldatp:// servername / pathNameonServer / databaseElement

`"aldatp://"` is the scheme of the URL that declares the type or protocol of the URL. "pathNameOnServer" gives the path name of the JRelix instance. "databaseElement" specifies a database resource or a task issued to the database. Relations, computations, views, relational expressions, statements and commands are examples of "databaseElement".

Both ALDATP and HTTP use a URL structure to identify and locate resource. The mechanism for ALDATP resembles that for HTTP in some aspects, but they are different.

- A resource on the Web is a corresponding file or code in a directory which the web server can access.
- A resource of the JRelix multi-database system is a corresponding database element of a JRelix instance running at a directory which other aldatp servers can access.
- A resource on the Web could be a static file.
- A resource of the JRelix multi-database system could be an existing relation, view, computation of a running JRelix instance.
- A resource on the Web could be a program (e.g. CGI, JSP) which Web server runs to dynamically generate a file on the fly and returns the file back to the client (browser).
- A resource of the JRelix multi-database system could be a relational expression which an aldatp server runs to dynamically generated a result relation on the fly and returns the relation back to the client; or it could be a statement block or command which an aldatp server runs and return a response code back to the client.

- For HTTP, some services-related codes could be downloaded to run inside the browser. For example, codes written in Javascript are often downloaded with a web form in order to provide better-quality interaction with the user.

- For ALDATP, top-level computations (like stored procedures in some databases) could be downloaded to run at the client site.

- On a web server only the files or codes published or put into directories that the web server can access are publicly available, others are private.

- In JRelix multi-database system, the JRelix running under a `"public_aldatp"` directory are publicly available, others are private.

- Web server can "point to" other Web servers with hypertext links. The file returned by the server normally contains pointers to other files that can reside on other servers. It is the client(browser) that open other TCP connections to follow these links from server to server to fetch the files.

- An aldat query issued by a client can nest distributed sub-queries. It is the Jrelix server that opens other TCP connections to follow these links from server to server.

The primary difference between ALDATP and HTTP is that ALDATP is designed for sharing resource among database systems while HTTP is for sharing resource among file systems.

Potentially, all the files and CGI or JSP codes put in the directories that the web server can access are publicly available via HTTP. On each host, one web server is enough to deal with the requests from clients (browsers). One web server can access all those files and invoke all those codes that are published.

In a JRelix multi-database system, a group of database sites might coexist in one host. Each database runs at an individual directory. Each directory running a JRelix instance could have a sub-directory in which another database instance is running there. A database tree is similar to a file system tree. The directory nodes in a file system tree correspond to the database nodes in a database tree. The files or codes in a directory correspond to the elements in a database. To access the elements of one database, that database must be running as a server. Unlike one web server for one host, there are a group of database servers for one host.

1.6 Outline of the Thesis

- Chapter 1 introduces the purpose and outline of this thesis. A literature rerview of distributed database management system, HTTP, URL and a database programming language JRelix is provided. The approach of this thesis is also briefly introduced in this chapter.
- Chapter 2 is a introduction to the JRelix on which our work is based.
- Chapter 3 is the users' manual on JRelix distributed data processing.
- Chapter 4 gives a detail description of implementing distributed data processing — Aldatp in JRelix.
- Chapter 5 presents possible extensions with Aldatp in JRelix. An application of event-based distributed system and suggestions to obtain location and fragemntation transparency are discussed.
- Chapter 6 concludes the thesis with a summary and proposes future works.

Chapter 2

JRelix System

This chapter presents a tutorial on JRelix so that readers will understand the rest of the thesis. This tutorial focuses on the parts of JRelix that are relevant to our implementation of multi-database systems.

2.1 Starting and Exiting JRelix

To start JRelix, the following command is typed on the command line of the operating system.

```
> java JRelix
```

As a result, JRelix copyright information is displayed in its run-time environment, and JRelix shows its prompt sign">" and waits for user input.

```
+-------------------------------------------+
|            Relix Java version 0.80        |
| Copyright (c) 1997 -- 2002 Aldat Lab      |
|         School of Computer Science        |
|             McGill University             |
+-------------------------------------------+
>
```

Figure 2.1: Initial Screen upon Starting JRelix

To exit the system, user types "quit;" after the system prompt sign.

2.2 Declaration

Domain Declaration

A relation is defined on one or more attributes, and the data for a given attribute is from a particular domain of values. The domian of a given attribute determines its data type. There are two kinds of domain declaration in JRelix, i.e. atomic-typed domain and complex-typed domain.

JRelix provides ten atomic data types: integer, short, long, float, double, boolean, string, text, universal [Mer01] and attribute [Mer01] type. In general, the syntax used to declare a domain of atomic data type is as follows:

```
>domain <dom_name1>,<dom_name2>... <data_type>;
```

The following are example of declaring atomic-typed domains.

```
> domain dept, office, code, name, title, address string;
> domain numStus integer;
```

On the other hand, two complex data types have been implemented in current JRelix, i.e. nested relation and computation. Nested relational domain is used when the attribute in a relation is a further relation. This mechanism constructs a nested relation. The syntax used to declare a nested relational domain is as follows:

```
> domain <nested_dom_name> (<dom_name1>,<dom_name2>,...);
```

The following are examples of declaring nested relational domain.

```
> domain course(code, title);
> domain profs(name, office, course);
```

When a new nested domain is declared, an invisible relation (whose name starts with a ".") is created automatically in the system. This relation is supposed to hold the data that belong to the nested domain in question. In the example given, ".course" and ".profs" are generated.

To show the information of domains currently declared in the system, use the following command:

```
> sd ;
> sd <dom_name>;
```

When domain name is specified, the command shows the information about this particular domain; otherwise, it shows all the currently declared domains.

The following is an example of showing all the domain lists.

```
>sd;
------------------------------- Domain Entry -------------------------------
Name        Type           NumRef     IsState   Dom_List
----------------------------------------------------------------------------
code        string         1          false
course      idlist         1          false     .id, code, title,
address     string         0          false
office      string         1          false
numStus     integer        0          false
dept        string         0          false
profs       idlist         0          false     .id, name, office, cours
title       string         1          false
name        string         1          false
----------------------------------------------------------------------------
>
```

To delete a domain from the current system, use the dd command:

```
> dd <dom_name>;
```

Declare and Initialize Relations

Relations are defined on one or more attributes (or domains) which must have been declared before the relation is declared or initialized. The general syntax for declaring a relation is as follows:

```
>relation <rel_name>(<dom_name1>, <dom_name2>...);
```

Note that the domain list can be any valid domains declared already in the system, e.g. atomic-data-typed domains, nested relational domains and computational domains etc.

The following are examples of relation declaration.

```
> relation student(dept, address, numStus);
> relation faculty(dept, profs);
```

However, the syntax given above only declares a relation structure in the system, which means it is an empty relation without any data inside. A relation can also be declared with actual data tuples, and this is called relation initialization. The syntax for relation initialization is defined as follows:

```
> relation <rel_name>(<dom_name1>,<dom_name2>...) <- <initialization_list>;
```

Figure 2.2 is a example of initalizing flat realtion and Figure 2.3 is a example of initalizing nested realtion.

```
department ( dept    address                  numStus)
             CS    3480 University Street      500
             MA    1001 Sherbrooke             800
             EE    3480 University Street      400

>relation department(dept, address, numStus) <- {
("CS", "3480 University Street", 500),
("MA", "1001 Sherbrooke", 800),
("EE", "3480 University Street",400)};
```

Figure 2.2: Initialize a flat relation: department

The rule for initialization list is

- A relation is always surrounded by a pair of curly brackets.
- Inside a relation, each tuple is surrounded by a pair of round brackets.
- Different tuples are separated by comma signs.

Although it seems only one relation (e.g. faculty) is initialized during a nested relation declaration, multiple relation initializations might potentially be involved. As mentioned before, when a nested domain is declared, a corresponding relation whose name is prefixed with a "." is created in the system automatically, and this relation is supposed to hold the data that belong to the nested domain. In the example given, .course and .profs are generated.

```
faculty
  ( dept      profs                                        )
           ( name   office    course                     )
                           (  code    title            )
   CS      Merrett  304      612     database system
                             617     information system
           Newborn  305      767     E-commerce
                             431     Algorithms
   EE      Pat      412      530     Control System
                             538     Robot

>relation faculty(dept, profs) <- {
("CS",  { ("Merrett","304",{ ( "612","database system"),
( "617", "information system")}), ( "Newborn", "305",
{ ( "767","E-commerce" ), ("431","Algorithms") } ) } ),

("EE",  { ("Pat","412", {  ( "530","Control system" ),
( "531", "Robot") } ) } )
};
```

Figure 2.3: Initialize a nested relation: faculty

The linkage between the top-level relation and relations associated with each nested domain is achived through a so-called "surrogate", which represented as a long integer in JRelix implementation. Figure 2.4 shows the surrogates linking relations.

To print the content of a relation in screen, use the **pr** command:

```
> pr <rel_name>;
```

The **dr** is used to remove the relation specified from the system:

```
>dr <rel_name>;
```

The command to list all relation entries that have been declared in the system is **sr**.

```
>sr;
>sr <rel_name>;
```

```
For example,
>sr;
-------------------- Relation Table ------------------------
Name             Type        Arity    NTuples   Sort    Ac
tive
---------------------------------------------------------------
```

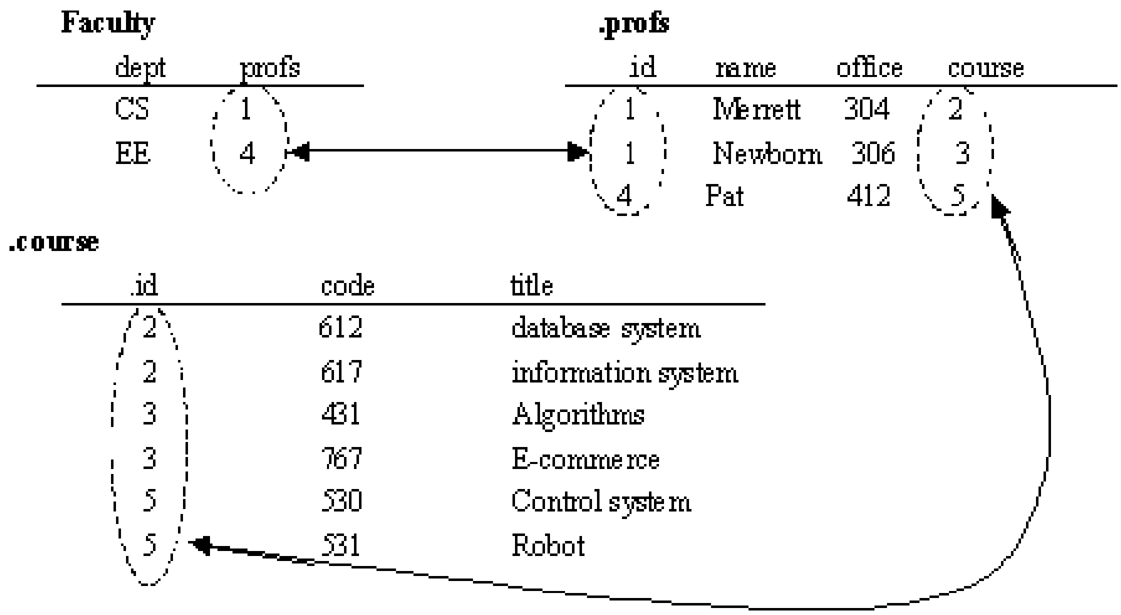


Figure 2.4: Link two relations through surrogates

```
department      relation       3         3         3      0
faculty         relation       2         2         2      0
--------------------------------------------------------------
>
```

2.3 Relational Algebra

Relational algebra consists of a set of functional operations on one or two relations and produces a result relation. JRelix constructs expressions by using various operators and then produces the result relation by assignment or incremental assignment.

Assignment and Incremental Assignment

An assignment `"<-"` creates a relation using the result of a relational expression. an incremental assignment `"<+"` adds the result of a relational expression to an existing relation. The general syntax is as follows:

```
> <new_relname> <- <expression>;
> <new_relname> <+ <expression>;
```

For assignment operation, if the result relation has the same name as an existing relation in the current system, the existing relation will be removed first.

```
Example
  > facultyCopy <- faculty;
  > department <+ newDepartment;
```

In the above examples, *facultyCopy* obtains a copy of original *faculty*. The result of *department* is a merge of the original *department* and *newDepartment*.

Relational Expression

Relational expression can be divided into two categories: unary operations and binary operations.

Unary operations

Unary operations take one relation as input and generate one relation as output. Projection, selection and T-selection are unary operations.

Projection

Projection creates a subset of the source relation specified by Expression. It extracts a subset of the attributes of the source relation by domain list. Duplicate tuples will be removed from the result relation. the syntax is as follows:

```
> [ <dom_name1>, <dom_name2>...] in <expression>;
```

Selection

Selection also creates a subset of the source relation specified by Expression. Unlike Projection, the result relation contains all the attributes of the source relation. However, the tuples in the result relation are those satisifying the condition of the SelectionClause. The syntax is as follows:

```
> where <SelectClause> in <Expression>;
```

T-selection

Projections and selections can be combined into one expression to form T-Selections. In a T-Selection, first perform the selection, and then perform the projection. The syntax is as follows:

```
> [<dom_name1>, <dom_name2>, ...] where <SelectClause> in <Expression>;
```

```
Example:
>R <- [dept] where numStus > 500 in department;
>pr R;
+----------------------+
| dept                 |
+----------------------+
| MA                   |
+----------------------+
relation R has 1 tuple
```

Binary Operations

Binary operations take two relations as input and produce one result relation. There are two categories of binary operators: mu-joins and sigma-joins. mu-joins are a generalization of set operations on relations, and sigma-joins are a generalization of logical operations on relations [Mer84]. The results of mu-joins and sigma-joins are also relations.

The syntax for join goes as follows:

```
<Expression> <JoinOperator> <Expression>
<Expression> [ <ExprList> : <JoinOperator> : <ExprList> ] <Expression>
```

In the first production, the common attributes of the left and right side relations are used as join attributes. In the case where the left and right side relations have no common attributes, the user may specify which attributes form the join attributes. This is handled in the second production.

μ-joins ("set"-valued)

μ-joins correspond to the binary set operations of union, intersection and difference. In general, μ-joins operators can be defined in terms of three components - center, left and right. Given two relations R(X,Y), S(Y,Z), the three components are defined as follows:

```
center(R,S) = {(x,y,z) |(x,y) in R and (y,z) in S}
```

```
left(R,S)   = {(x,y,dc)|(x,y) in R and any z ((y,z) not in S)}
right(R,S)  = {(dc,y,z)|(y,z) in S and any x ((x,y) not in R)}
```

```
We have:
R ujoin S = center(R,S) U left(R,S) U right(R,S)
R ijoin S = center(R,S)
R djoin S = X,Y in left(R,S)
R drjoin S = Y,Z in right(R,S)
R lrjoin S = left(R,S) U center(R,S)
R rjoin S = right(R,S) U center(R,S)
R sjoin S = left(R,S) U right(R,S)
```

Take the two relations "department" and "faculty" introduced in Figure 2.2 and Figure 2.4 as examples.

```
>R <- department ijoin faculty;
>pr R;
+---------+---------------------+------------+----------+
| dept    | address             | numStus    | profs    |
+---------+---------------------+------------+----------+
| CS      | 3480 University Stre | 500       | 1        |
| EE      | 3480 University Stre | 400       | 4        |
+---------+---------------------+------------+----------+
relation R has 2 tuples

>R <- department ujoin faculty;
>pr R;
+---------+---------------------+------------+------------+
| dept    | address             | numStus    | profs      |
+---------+---------------------+------------+------------+
| CS      | 3480 University Stre | 500       | 1          |
| EE      | 3480 University Stre | 400       | 4          |
| MA      | 1001 Sherbrooke     | 800        | dc         |
+---------+---------------------+------------+------------+
relation R has 3 tuples
Note: dc means "don't care".

> R <- department djoin faculty;
>pr R;
+---------------------+---------------------+------------+
| dept                | address             | numStus    |
+---------------------+---------------------+------------+
| MA                  | 1001 Sherbrooke     | 800        |
```

```
+---------------------+---------------------+-------------+
relation R has 1 tuple
```

σ-joins ("truth"-valued)

The family of σ-joins are based on set comparison operators such as "subset" or "equals". The sigma joins extend the truth-valued comparison operation on sets to relations by applying them on each set of values of the join attribute for each of the other values in the two relations. Refer to [Mer84] for detail descriptions on σ-joins.

One of the frequently used sigma-joins is natural composition, i.e. icomp. The operations and result of icomp are quite similar to that of natural join (i.e. ijoin), except that the join attributes are removed from the result relation.

```
Example
>R <- department icomp faculty;
>pr R;
+---------------------+-------------+-----------------+
| address             | numStus     | profs           |
+---------------------+-------------+-----------------+
| 3480 University Stre | 400        | 4               |
| 3480 University Stre | 500        | 1               |
+---------------------+-------------+-----------------+
relation R has 2 tuples
```

2.4 Domain Algebra

Domain algebra provides a set of operations applied on attributes. A thorough description of domain algebra can be found in [Mer84]. Although domain algebra is one of the most important components for JRelix, we are not going to elaborate it here because it is not very crucial to our implementation of multi-database system. For further information, please refer to [Mer84, Yua98].

2.5 Views

While the assignment operator causes the expression following it to be evaluated and the result stored in the relation named on the left, it is useful to be able to defer the evaluation until later. The mechanism for this is called a view. Unlike a relation, a view does not hold data upon declaration and initialization. It is usually regarded as a functional definition. In JRelix notation, **is** replaces the assignment arrows, `<-` and `<+`. Thus the syntax for views is as follows:

```
> <view-name> is <expression> ;
```

It defines view-name to be synonymous with the result relation of the relational expression, and no evaluation is performed until a subsequent assignment, or other operation such as print forces it.

Example

```
>V is faculty sjoin department;
>pr V;
+-------------------+-------------------+-------------------+----------+
| dept              | profs             | address           | numStus  |
+-------------------+-------------------+-------------------+----------+
| MA                | dc                | 1001 Sherbrooke   | 800      |
+-------------------+-------------------+-------------------+----------+
expression has 1 tuple
```

2.6 Update

The update operation allows us to change values of specified attributes in certain tuples. These attributes could be selected by a "using" clause which uses a relational algebra operation to select tuples from the relation we want to update. We can also use updates to add or delete some tuples to or from the relation.

Update provides the mechanism for changing a relation. There are three basic update operations on relations: *add, delete and change.* The syntax for update is:

```
>Update <rel_name> add <expression>;
>Update <rel_name> delete <expression>;
>Update <rel_name> change <statementList> <UsingClause >;
UsingClause := using <JoinOperator> <Expression>
```

Here the first two productions add or delete the result of Expression to or from the relation being updated. The semantics of add is the same as that of the incremental assignment. The semantics of delete is related to that of the djoin. The third production updates part of the relation in the way specified by StatementList. The part of the relation to be updated is the join result (specified by JoinOperator) of the relation being updated and the result of expression in UsingClause. If there is no JoinOperator in the UsingClause, the default join operator is natural join. If there is no UsingClause, the whole relation is updated. The StatementList that follows the keyword change may contain update statements.

Example

```
>update department change numStus <- numStus+100 using ijoin on faculty;
>pr department;
+----------------------+----------------------+-------------+
| dept                 | address              | numStus     |
+----------------------+----------------------+-------------+
| CS                   | 3480 University Stre | 600         |
| EE                   | 3480 University Stre | 500         |
| MA                   | 1001 Sherbrooke      | 800         |
+----------------------+----------------------+-------------+
relation department has 3 tuples
```

2.7 Computations

Computation implements procedural abstraction in jRelix. The basis of computation can be found in [Mer84]. The formal syntax for the declaration of a computation goes as follow:

```
comp <comp_name> ( < ParameterList > )is
< ComputationBody > ;
```

A computation can be thought of as a compressed relation, in which the relationship is given not explicitly by data but implicitly by code. For example *IntPerChg* is a relationship among *I, i, p.*

```
>domain I,i float;
>domain p integer;
>comp IntPerChg(I,i,p) is
{I <- (1+i)**p-1}
alt
{i <- (1+I)**(1.0/p)-1}
alt
{p <- round(log(1+I)/log(1+i))};
```

The computation name is *IntPerChg*. There are three parameters in this computation, and they are all defined as domains. There are three "alt" blocks in this example. All of them satisfy the constraint "I = (1+i)**p-1". Given values for any two of these variables, the value of the third will be calculated according to the constraint.

The central design principle applied in implementing computation is to make them resemble relations. We show the relation corresponding to velocity as below:

```
IntPerChg ( I          i                 p  )
            0.06    0.0024307966       24
            0.06    0.0048675537       12
            0.07    0.0028231144       24
            0.07    0.0056540966       12
            ......
```

It is an infinite relation, in which every tuple satisfies the constraint "I = (1+i)**p-1". Further more, all tuples satisfying this constraint are included in the relation. The parameters of a computation become the domains of its associated relation.

Although IntPerchg is code and we call it a computation or comp, it must always be thought of as a relation. That way, we do not need any new syntax to invoke it. For example,

```
> Intint <- [p] where I=0.12 & i=0.01 in IntPerChg;
gives the result as follow.
>pr Intint;
+-------------+
| p           |
+-------------+
| 11          |
+-------------+
```

Computations may also be invoked by means of top-level calls in jRelix taking domains and relations as parameters or taking no parameters. Thus computation functions just as procedure, which is a top level procedural abstraction implemented in Relix, the predecessor of jRelix. Please refer to [RSL95] for a complete discussion of procedure in Relix. Take the following example:

```
> comp AssignComp ( ) is
   { result <- temprel;
   };
> comp JoinComp ( A, B, C ) is
   { C <- A ujoin B;
   };
> AssignComp ( );
> JoinComp ( in oldRecord, in moreRecord, out Record);
```

Here we have defined two computations. The invocation of computation *AssignComp* is a top level call which takes no parameter. The invocation of computation *JoinComp* is also a top level call, taking three parameters where in and out are used to specify the input and output parameters. "oldRecord" and "moreRecord" are two existing relations, while "Record" is the output and newly generated relation in this example. The statement(s) in the computation body will be executed when the computation is called.

2.8 Event Handlers

An event is a system-generated procedure call. The procedure that is called is commonly known as an event handler. It is a procedure with a specially formulated name linking it to the situation under which it should be called. For example, when a relation is updated by *add, delete or change,* the response can be coded in procedures named, respectively, with the following syntax:

```
[pre|post:] add: <relation>
[pre|post:] delete: <relation>
[pre|post:] change: <relation>: <attribute>
```

where *pre* means that the procedure is to be invoked before executing the update, and *post* means call the procedure afterwards.

For the update operation, the affected relation would be separated into three pieces which are named Trigger, New and Rest. They only exist in the pre and post event handler. Trigger is defined as being the tuples which will be affected by an update operation on the original relation, or you can call it Old. New is defined as being the new values of those old tuples. Rest is defined as being the tuples which are not affected by an update operation on the original relation. They only exist in the pre and post event handler.

2.9 System Commands

System commands can be used to set JRelix environment and display system information. By using these commands, the user can know more about his or her environment upon starting JRelix run-time system.

The following are some of the system commands implemented in JRelix.

Debug: turn the debug model on/off

Time: turn on/off the interpretation timer.

Trace : turn the log on/off

Sd: display the user-defined and system-defined domain information.

Sr: display the user-defined and system-defined relation information

Input: any JRelix commands and statements can be stored as a batch file on the disk and be loaded into the system like a sequence of JRelix commands

Chapter 3

Users' Manual on JRelix Distributed Systems

In this chapter, the users' manual on remote data processing of JRelix is given. Section 3.1 describes a demo environment used throughout this user manual, and introduces basic ways to launch the multiple JRelix system. Section 3.2 presents the new syntax for visiting other databases. Sections starting from 3.3 to 3.9 go through the detailed examples about the remote data processing of this system. The topics cover remote assignment, view, update, computation, computation call, statement block and command. In section 3.10, we show four basic options to start JRelix. There are extensions that will be introduced in other sections. Section 3.11 describes the way to manage the port numbers used by the multiple JRelix systems. Section 3.12 introduces background servers that will be started and stopped automatically. In Section 3.13, security issues on file access permission and the protected server running outside "public_aldatp" directories are illustrated.

3.1 Getting Started

Before starting the multiple JRelix, we must illustrate a demo environment in our lab first. Throughout this chapter, we use examples on this environment to show the way to start and use this system. Figure 3.1 presents the structure of the demo multiple JRelix system.

Two physical hosts, "roo" and "mimi", are involved in the example. "~tim", and "~zwang26", are two normal users' home directories respectively. Figure 3.1 shows "zwang26's" and "tim's" home directories on "roo", and only "zwang26's" home directory on "mimi".

Immediately under each user's home directory, there is a "public_aldatp" directory. Each node of the sub-tree rooted at "public_aldatp" is an individual JRelix. For instance, under "~zwang26/" on "roo", "public_aldatp", "pubA", "pubA1", "pubA2", "pubB" are all individual JRelix systems. Under normal conditions, the JRelix running under "public_aldatp" directory is publicly available. It could be visited by other databases that are potentially linked by Internet. As is illustrated in the Figure 3.1, except under a regular user's home directory, "public_aldatp" may also reside at a higher level directory, e.g. its owner is "root" .

Besides the nodes under a "public_aldatp", JRelix could also run at a site out of the "public_aldatp". In Figure 3.1, The JRelix running at "~zwang26/JRelix/priv" on "roo" and the JRelix running at "~tim/priv" on "roo" are such sorts of systems. By default, the JRelix running outside "public_aldatp" is private and protected. Only the local owner can operate it. However, it is able to visit other publicly available JRelix. For instance, the JRelix running at "~tim/priv" can visit other JRelix running under "public_aldatp".

The basic way to start a multiple JRelix system could be very simple. In Section 3.10, more options and situations will be introduced in detail.

Step 1. Start JRelix Root Server

On each machine, **one and only one** root server should be launched before the whole multiple JRelix system can work.

The DBA or system administrator with super user authority, goes to the root user's "public_aldatp" directory (the "public_aldatp" at the upper right corner on Fig 3.1), and types "`java aldatpTLd &`".

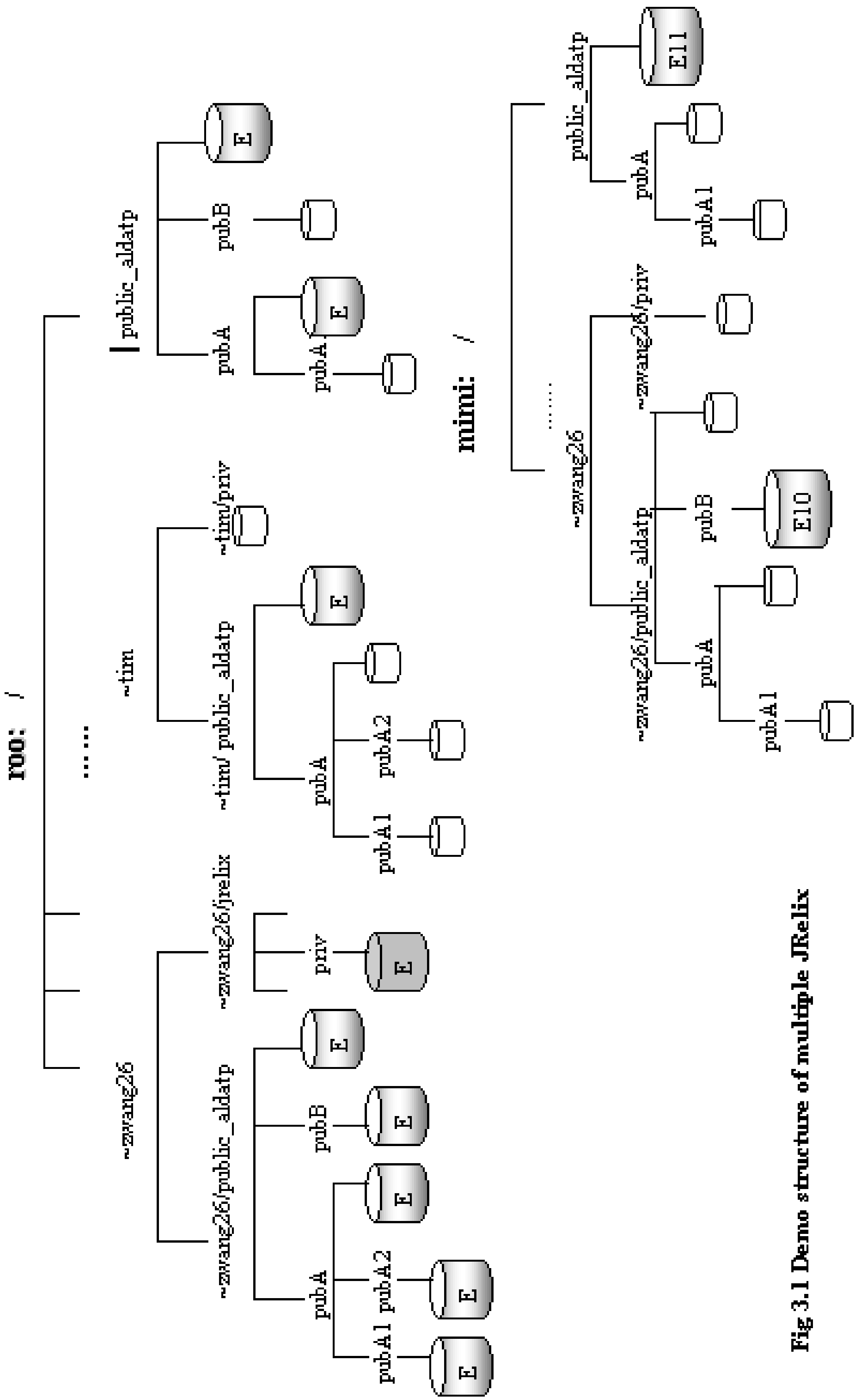


Figure 3.1: Demo JRelix Systems

Or

The aldatpTLd root server can also be launched by a common user. For normal user, go to the "public_aldatp" directory, which should be immediately under his home directory, and type "`java aldatpTLd &`"

Example 3.1.1

```
[zwang26][roo][~/public_aldatp] java aldatpTLd &
```

Note:

If the aldatpTLd root server is launched by a common user, the "public_aldatp" at the root level directory is publicly unavailable. More descriptions are given in section 3.10.

From this section to section 3.9, it is assumed that the root level server aldatpTLd is launched at the root user's "public_aldatp" instead of at a normal user's "public_aldatp". So the root user level "public_aldatp" is publicly available in the examples of these sections.

Step 2. Start JRelix Instance

Go to the directory of one JRelix database, and type "java JRelix"

Example 3.1.2

```
[zwang26][roo][~/public_aldatp/pubA] java JRelix
Starting lower-level aldatp server using port:9994
+--------------------------------------------+
|            Relix Java version 0.80         |
| Copyright (c) 1997 -- 2002 Aldat Lab       |
|         School of Computer Science         |
|             McGill University              |
+--------------------------------------------+
>
```

3.2 Syntax of aldatp

To enable two JRelix to visit each other, a communication must be established between them so that shipping data, shipping query, and executing remote command is realizable. To implement these communication abilities, new protocols are developed which are collectively called aldatp.

We are not going to explain these protocols in this user's manual. In this section, we are going to introduce how to use the new syntax for aldatp to express the intentions of shipping data, shipping queries, executing remote commands and remote computation calls. From section 3.3 to section 3.9, detailied tutorials are given on using them.

To visit other JRelix, we need to indicate the location of the destination. The location information about a JRelix system includes the host name or IP address of the physical machine and its position in the file system. So host name or IP address plus the path or directory in the file system make a global unique identifier for each JRelix on the Internet.

Eleven sample databases are highlighted in Figure 3.1. On each sample database, a representive element labeled Ei (i: from 1 to 11) stands for the elements of the corresponding database. An element in our example could be one of the followings:

1. name of a relation, view or computation,
2. parenthesized expression,
3. statement or statement block in brace.

Table 3.1 presents the syntax for accessing these E1 to E11 from each 11 databases.

The following is a example indicating the element E1 of JRelix running at host "roo" on the directory *~zwang26/public_aldatp/pubA/pubA1*

Example 3.2.1

aldatp://roo/~zwang26/pubA/pubA1/E1

As it can be seen, the syntax for remote relation identifier

1. begins with "aldatp://", which is a header, followed by
2. the host name "roo",
3. alias of user home directory "~zwang26/",
4. part of the path "pubA/pubA1/",
5. ends with the relation name "E1".

As mentioned in section 3.1, not all the JRelix on the Internet are designed to be publicly available. However, by default, the JRelix under "public_aldatp" are publicly

Location	**Syntax for accessing E1 to E11**
DB 1	E1 aldatp://localhost/~zwang26/pubA/pubA2/E2 aldatp://localhost/~zwang26/pubA/E3 aldatp://localhost/~zwang26/pubB/E4 aldatp://localhost/~zwang26/E5 E6 not available aldatp://localhost/~tim/E7 aldatp://localhost/pubA/E8 aldatp://localhost/E9 aldatp://mimi/~zwang26/pubB/E10 aldatp://mimi/E11 **Note:** "localhost" and "roo" are alternative to each other for site 1
DB 2	aldatp://localhost/~zwang26/pubA/pubA1/E1 E2 From E3 to E11, the same as site1 **Note:** "localhost" and "roo" are alternative to each other for site 2
DB 3	PubA1/E1 or aldatp://localhost/~zwang26/pubA/pubA1/E1 PubA2/E2 or aldatp://localhost/~zwang26/pubA/pubA2/E2 E3 From E4 to E11, the same as site1 **Note:** "localhost" and "roo" are alternative to each other for site 3
DB 4	aldatp://localhost/~zwang26/pubA/pubA1/E1 aldatp://localhost/~zwang26/pubA/pubA2/E2 aldatp://localhost/~zwang26/pubA/E3 E4 From E5 to E11, the same as site1 **Note:** "localhost" and "roo" are alternative to each other for site 4
DB 5	PubA/pubA1/E1 or aldatp://localhost/~zwang26/pubA/pubA1/E1 PubA/pubA2/E2 or aldatp://localhost/~zwang26/pubA/pubA2/E2 PubA/E3 or aldatp://localhost/~zwang26/pubA/E3 PubB/E4 or aldatp://localhost/~zwang26/pubB/E4 E5 From E6 to E11, the same as site1 **Note:** "localhost" and "roo" are alternative to each other for site 5

Location	**Syntax for accessing E1 to E11**
DB 6	aldatp://localhost/˜zwang26/pubA/pubA1/E1 From E2 to E5 , the same as site1 E6 From E7 to E11, the same as site 1 **Note:** "localhost" and "roo" are alternative to each other for site 6
DB 7	aldatp://localhost/˜zwang26/pubA/pubA1/E1 From E2 to E6 , the same as site1 E7 From E8 to E11 , the same as site1 **Note:** "localhost" and "roo" are alternative to each other for site 7
DB 8	aldatp://localhost/˜zwang26/pubA/pubA1/E1 From E2 to E7, the same as site1 E8 From E9 to E11 , the same as site1 **Note:** "localhost" and "roo" are alternative to each other for site 8
DB 9	aldatp://localhost/˜zwang26/pubA/pubA1/E1 From E2 to E7 , the same as site1 PubA/E8 or aldatp://localhost/pubA/E8 E9 From E10 to E11 , the same as site1 **Note:** "localhost" and "roo" are alternative to each other for site 9
DB 10	aldatp://roo/˜zwang26/pubA/pubA1/E1 aldatp://roo/˜zwang26/pubA/pubA2/E2 aldatp://roo/˜zwang26/pubA/E3 aldatp://roo/˜zwang26/pubB/E4 aldatp://roo/˜zwang26/E5 E6 not available aldatp://roo/˜tim/E7 aldatp://roo/pubA/E8 aldatp://roo/E9 E10 aldatp:/localhost/E11 **Note:** "localhost" and "mimi" are alternative to each other for site 10
DB 11	From E1 to E9, the same as site 10 Pub/E10 or aldatp://loclahost/˜zwang26/pub/E10 E11 **Note:** "localhost" and "mimi" are alternative to each other for site 11

Table 3.1: URL-based names.

available and those outside "public_aldatp" are private and protected. Therefore, we do not use the whole path in our syntax. Instead, we only use the part of the path which immediately after "public_aldatp" to represent the path information. By using this structure, there are two apparent advantages. First, outside "public_aldatp" JRelix are protected since there are no ways to parse a directory outside "public_aldatp". Second, the users are not required to know the whole complicated path. The knowledge of the structure of the sub tree rooted at "public_aldatp" is enough.

Example 3.2.2

1) aldatp://roo/E5

Element E5 of the JRelix running on "roo" at "~/public_aldatp"

2) aldatp://mimi/pubB/pub/E10

Element E10 of the JRelix running on "mimi" at "~/public_aldatp/pubB"

3) aldatp://localhost/~tim/E7

Element E7 of the JRelix running on local host at "~tim/public_aldatp"

There is a shortcut for this expression. But it only applies to a parent JRelix accessing its descendants

Example 3.2.3

For JRelix running on "roo" at "~tim/public_aldatp/pubA" , if it accessies E1 of its descendant JRelix running on "roo" at "~tim/public_aldatp/pubA/pubA1", a shortcut exists:

```
pubA1/E1
```

This shortcut is only suitable for "going down". Any "going up", or first "going up" then "going down" can not use this format.

Example 3.2.4

For the JRelix running on "roo:~zwang26/public_aldatp/pubA" , if it wants to access E4 M of "roo:~zwang26/public_aldatp/pubB", the following expression must be used:

aldatp://roo/~zwang26/pubB/E4

If it use pubB/E4, the system will look for "~zwang26/public_aldatp/pubA/pubB" which does not exist.

The syntax for remote views and computations are the same as for relations. With the knowledge of the syntax for identifying remote relations, views or computations, it is easy to understand the syntax for remote expressions and statements. The syntax for remote expressions is to replace the identifier (for relation, view, or computation) with a parenthesized expression.

Example 3.2.5

aldatp://roo/pubA/(R1 ijoin S1)

aldatp://localhost/~tim/([name] in R7 ijoin aldatp://mimi/R11)

Similarly, the syntax for remote statements is to replace the identifiers with braced statements or statement blocks.

Example 3.2.6

```
aldatp://roo/pubA/{ let name' be name};

aldatp://mimi/~zwang26/pubB/{ S4 <- R4 };

aldatp://roo/~tim/pubA/pubA1/{ update R add S };

aldatp://localhost/~tim/pubA/{ compcall (in R, out aldatp://mimi/pubA/R) };
```

More examples on declarations, commands, assignments, updates, views, computations, computation calls, and statement blocks will be given in the following sections.

3.3 Remote Assignment

Note: Suppose all the example databases are empty at the beginning of this section. This section is a self-contained tutorial. Users can follow the tutorial step by step.

As we know, in JRelix system, the domain must be defined before it can be used to create new relations. For instance, to create the following relation:

```
Students3 ( name    date     )
            Joe     09/2001
            Sue     09/2002
```

We need to define the domains first if they do not already exist in the system.

Example 3.3.1

1) start JRelix at "roo:~zwang26/public_aldatp/pubA";

```
[zwang26][roo][~/public_aldatp/pubA] java JRelix
Starting lower-level aldatp server using port:9994
+------------------------------------------+
|            Relix Java version 0.80       |
| Copyright (c) 1997 -- 2002 Aldat Lab     |
|         School of Computer Science       |
|             McGill University            |
+------------------------------------------+
```

2) define domains "name", "date" and initialize relation "Students3"

```
>domain name,date string;
>relation Students3(name,date) <- {("Joe","09/2001"),("Sue","09/2002")};
>pr Students3;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| Joe                  | 09/2001              |
| Sue                  | 09/2002              |
+----------------------+----------------------+
relation Students3 has 2 tuples
>
```

Now suppose we will assign this "Students3" relation from "roo:~zwang26/ public_aldatp/pubA" to the "roo:~zwang26/JRelix/priv". There are two equal ways to do it; either do the assignment at "roo:~zwang26/JRelix/priv", or do the assignment at "roo:~zwang26/public_aldatp/pubA". Before we do that, we might be concerned about whether the domains "name" and "date" have been defined at "roo:~zwang26/JRelix/priv" or not.

For the "Students3" example, if any of the "name" and "date" is not defined at the "roo:~zwang26/JRelix/priv", the system would automatically define a new one at "roo:~zwang26/JRelix/priv" exactly the same as what is defined at "roo:~zwang26/

public_aldatp/pubA". If user would like, he could define "name" or "date" manually, but this is not necessary.

Example 3.3.2

1)Start JRelix at "roo:~zwang26/JRelix/priv", display the domain list existing in the system. In this example, it is empty.

```
[zwang26][roo][~/JRelix/priv] java JRelix
Starting protected lower-level aldatp server using port:9993
......  //version info
>sd;
---------------------- Domain Entry -------------------------------
Name        Type            NumRef     IsState   Dom_List
-------------------------------------------------------------------

2) Do the assignment and check the domains again.
>Stu6A <- aldatp://localhost/~zwang26/pubA/Students3;
>sd;
---------------------- Domain Entry -------------------------------
Name        Type            NumRef     IsState   Dom_List
-------------------------------------------------------------------
date        string          1          false
name        string          1          false
-------------------------------------------------------------------
>pr Stu6A;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| Joe                  | 09/2001              |
| Sue                  | 09/2002              |
+----------------------+----------------------+
relation Stu6A has 2 tuples
>
```

If at "roo:~zwang26/JRelix/priv", "name" or "date" are defined, but have different type, then the system would check if any relation refers to the domain. If it were being used, the system would complain "conflict domain detected", or else replace it with the new one.

Example 3.3.3

1) go to JRelix running at "roo:~zwang26/JRelix/priv", delete relation "Stu6A", domain "name" and "date"

```
>dr Stu6A;
>dd date,name;
```

2)define domain "name" and "date" again, but different type

```
>domain name,date real;
>sd;
---------------------- Domain Entry -------------------------------
Name        Type            NumRef     IsState   Dom_List
-------------------------------------------------------------------
date        float           0          false
name        float           0          false
-------------------------------------------------------------------
>
```

3)do the same assignment again and check the domain list.

```
>Stu6A <- aldatp://localhost/~zwang26/pubA/Students3;
>sd;
---------------------- Domain Entry -------------------------------
Name        Type            NumRef     IsState   Dom_List
-------------------------------------------------------------------
date        string          1          false
name        string          1          false
-------------------------------------------------------------------
>
```

Note: Although the domains have the "same name, different type", they are replaced with new ones since they are not being used.

4)Delete relation "Stu6A", domains "name" and "date". Define new "name" and "date" using the different type, and create a relation using these domains. Do the assignment again.

```
>dr Stu6A;
>dd name,date;
>domain name,date real;
>relation R(name,date) <- {(1.0,2.0),(3.0,4.0)};
>sd;
---------------------- Domain Entry -------------------------------
Name        Type            NumRef     IsState   Dom_List
-------------------------------------------------------------------
date        float           1          false
name        float           1          false
-------------------------------------------------------------------
>Stu6A <- aldatp://localhost/~zwang26/pubA/Students3;
InterpretError:  Conflict domain : name is being used as real
```

The last possibility is "same name, same type". In this case, only the number of reference is changed.

Example 3.3.4

1) go to JRelix running at "roo:~zwang26/JRelix/priv". Delete relation "Stu6A", domains "name" and "date".

```
>dr R;
>dd name,date;
```

2) assign to relation "Stu6A"

```
>Stu6A <- aldatp://localhost/~zwang26/pubA/Students3;
```

3) assign to another relation "Stu6B"

```
>Stu6B <- aldatp://localhost/~zwang26/pubA/Students3;
```

4) Show the domain list. The number of reference is increased to 2.

```
>sd;
---------------------- Domain Entry -------------------------------
Name        Type           NumRef     IsState   Dom_List
-------------------------------------------------------------------
date        string         2          false
name        string         2          false
```

The remote relation could appear on the right-hand side, the left-hand side or both sides of the assignment. The assignments could be a replacement one or an incremental one.

Example 3.3.5

1) go to "roo:~zwang26/public_aldatp/pubA", update Students3;

```
>update Students3 change name <- "aaa";
```

2) go to "roo:~zwang26/JRelix/priv", do the following assignment

```
>aldatp://localhost/~zwang26/Students5 <-
          aldatp://localhost/~zwang26/pubA/Students3 ujoin Stu6B;
OK
>
```

Example 3.3.6

1) go to "roo:~zwang26/JRelix/priv", incremental assign "Students3" at "roo:~zwang26/public_aldatp/pubA" to "Stu6A"

```
>Stu6A <+ aldatp://localhost/~zwang26/pubA/Students3;
>pr Stu6A;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| Joe                  | 09/2001              |
| Sue                  | 09/2002              |
| aaa                  | 09/2001              |
| aaa                  | 09/2002              |
+----------------------+----------------------+
relation Stu6A has 4 tuples
>
```

The JRelix system supports relational algebra and domain algebra for relations with an arbitrary level of nesting. The following example presents the way to ship nested relations.

Here is the nested relation we used in the example.

```
Books4
(  Authors   title    price    Descriptors  )
  (authors)                   (descriptors)
    A1       T1       P3         D1
    A2                           D2

    A1       T3       P1         D1
                                 D2
                                 D3
```

Example 3.3.7

1)Go to "roo:~zwang26/public_aldatp/pubB", which has a nested relation Books4;

```
> pr Books4;
+-----------------+-----------------+-----------------+-----------------+
| Authors         | title           | price           | Descriptors     |
+-----------------+-----------------+-----------------+-----------------+
| 3               | T3              | P3              | 4               |
| 1               | T1              | P1              | 2               |
+-----------------+-----------------+-----------------+-----------------+
relation Books4 has 2 tuples
```

```
>pr .Authors;
+---------------------+---------------------+
| .id                 | authors             |
+---------------------+---------------------+
| 1                   | A1                  |
| 1                   | A2                  |
| 3                   | A1                  |
+---------------------+---------------------+
relation .Authors has 3 tuples
>pr .Descriptors;
+---------------------+---------------------+
| .id                 | descriptors         |
+---------------------+---------------------+
| 2                   | D1                  |
| 2                   | D2                  |
| 4                   | D1                  |
| 4                   | D2                  |
| 4                   | D3                  |
+---------------------+---------------------+
relation .Descriptors has 5 tuples
>
```

2) assign Books4 to "roo:~zwang26/public_aldatp/pubA",

```
>aldatp://localhost/~zwang26/pubA/Books3 <- Books4;
```

3)go to JRelix running on "roo:~zwang26/public_aldatp/pubA", check the "Books3"

```
>pr Books3;
+------------------+------------------+------------------+------------------+
| Authors          | title            | price            | Descriptors      |
+------------------+------------------+------------------+------------------+
| 1                | T3               | P3               | 3                |
| 2                | T1               | P1               | 4                |
+------------------+------------------+------------------+------------------+
relation Books3 has 2 tuples
>pr .Authors;
+---------------------+---------------------+
| .id                 | authors             |
+---------------------+---------------------+
| 1                   | A1                  |
| 2                   | A1                  |
| 2                   | A2                  |
+---------------------+---------------------+
relation .Authors has 3 tuples
>pr .Descriptors;
+---------------------+---------------------+
| .id                 | descriptors         |
+---------------------+---------------------+
| 3                   | D1                  |
| 3                   | D2                  |
```

```
| 3                    | D3                   |
| 4                    | D1                   |
| 4                    | D2                   |
+----------------------+----------------------+
relation .Descriptors has 5 tuples
>
```

Note: Each system has its own sequential series for surrogates. When "Books4" is shipped from "roo:~zwang26/public_aldatp/pubB" to "roo:~zwang26/public_aldatp/pubA", new surrogates are allocated at "roo:~zwang26/public_aldatp/pubA".

3.4 Remote View

The assignment operator causes the expression following it to be evaluated and the result stored in the relation named on the left. For view, the evaluation is performed until a subsequent assignment, or other operation such as print, forces it.

The following example is about remote view. The view is defined for other JRelix, and the right-hand side expression could be local, remote or hybrid. Like the relation, this view can be cited at any site.

Example 3.4.1

1) go to JRelix at "roo:~zwang26/JRelix/priv", define the following view

```
>aldatp://localhost/~zwang26/StudentsView6 is Stu6B
          ujoin aldatp://localhost/~zwang26/pubA/Students3;
OK
```

2) print the view "StudentsView6" at "roo:~zwang26/JRelix/priv"

```
>pr aldatp://localhost/~zwang26/StudentsView6;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| Joe                  | 09/2001              |
| Sue                  | 09/2002              |
| aaa                  | 09/2001              |
| aaa                  | 09/2002              |
+----------------------+----------------------+
relation _temp_X9X_7 has 4 tuples
>
```

3)print the view "StudentsView6" at "roo:~zwang26/public_aldatp"

```
>pr StudentsView6;
......
>
```

Recursive remote view is also supported.

Example 3.4.2

Suppose relation "Parent6" exists at "roo:~zwang26/JRelix/priv"

```
Parent6( Sr      Jr   )
         Joe     Sue
         Max     Ann
         Max     Ted
         Sue     Max
```

The "Ancestor3" is a remote recursive view.

```
>aldatp://localhost/~zwang26/pubA/Ancestor3 is
   Parent6 ujoin (Parent6[ Jr : icomp : Sr] aldatp://localhost/~zwang26/pubA/Ancestor3);
```

If we print it, the evaluation is performed and the result is as follows

```
>pr aldatp://localhost/~zwang26/pubA/Ancestor3;
+----------------------+----------------------+
| Sr                   | Jr                   |
+----------------------+----------------------+
| Joe                  | Ann                  |
| Joe                  | Max                  |
| Joe                  | Sue                  |
| Joe                  | Ted                  |
| Max                  | Ann                  |
| Max                  | Ted                  |
| Sue                  | Ann                  |
| Sue                  | Max                  |
| Sue                  | Ted                  |
+----------------------+----------------------+
relation _temp_X9X_15 has 9 tuples
>
```

3.5 Remote Update

The updated relation could be a local or remote one. The right-hand expression could be local, remote or hybrid.

Example 3.5.1

1) Update local relation using remote expression

```
>update Stu6B add aldatp://localhost/~zwang26/pubA/Students3;

>update Stu6B delete aldatp://localhost/~zwang26/pubA/Students3;

>update Stu6B change name <- "Joe Joe" using
```

2)Update remote relation

```
>update aldatp://localhost/~zwang26/pubA/Students3 add Stu6B;
OK
>update aldatp://localhost/~zwang26/pubA/Students3
    change name <- if date="09/2002" then "Sue" else "Joe";
OK
```

3.6 Remote Computation

In JRelix, computation is a special kind of relation. Although it actually works as procedures or functions, it behaves like relations. Because a computation can be thought of as a typed relation, we do not need to add new syntax to invoke it. We are already quite familiar with remote relation and view, therefore it would be easy to work with remote computation.

Example 3.6.1

1) go to JRelix running at "roo:~zwang26/public_aldatp/pubA", create the following computation:

```
> domain I,i real;
> domain p integer;
>comp IntPerChg3(I,i,p) is
{I <- (1+i)**p-1}
alt
{i <- (1+I)**(1.0/p)-1}
alt
{p <- round(log(1+I)/log(1+i))};
>
```

2) go to JRelix at "roo:~zwang26/JRelix/priv" to invoke the computation "IntPerChg" at "roo:~zwang26/public_aldatp/pubA"

```
>Intint6 <- [p] where I=0.12 & i=0.01 in aldatp://localhost/~zwang26/pubA/IntPerChg3;
>pr Intint6;
+-------------+
| p           |
+-------------+
| 11          |
+-------------+
relation Intint6 has 1 tuple
>
```

3.7 Remote Computation Call

Computation calls are top-level computations. They could take relations as parameters, and these *in* or *out* parameter relations could be remote or local ones. The computation call itself could also be executed locally or remotely.

Example 3.7.1

1)The following is a simple example of computation call defined at "roo:~zwang26/public_aldatp/pubA"

```
> domain R(name,date);
> domain T(name,date);
> comp compcall3(R,T) is
{R <- T}
alt
{T <-R };
```

2)Invoke "compcall3". the "IN" parameter is a local relation and the "OUT" parameter is a remote relation.

```
>compcall3 (in Students3, out aldatp://localhost/~zwang26/Stud5);
>pr aldatp://localhost/~zwang26/Stud5;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| aaa                  | 09/2001              |
| aaa                  | 09/2002              |
+----------------------+----------------------+
relation _temp_X9X_146 has 2 tuples
```

Example 3.7.2

1) go to JRelix running at "roo:~zwang26/JRelix/priv", invoke "compcall3" at "roo:~zwang26/public_aldatp/pubA", both of the "IN" and "OUT" parameter relations are located at "roo:~zwang26/public_aldatp/pubA"

```
> aldatp://localhost/~zwang26/pubA/{compcall3 (in Students3, out R3)};
OK
>pr aldatp://localhost/~zwang26/pubA/R3;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| aaa                  | 09/2001              |
| aaa                  | 09/2002              |
+----------------------+----------------------+
relation _temp_X9X_38 has 2 tuples
>
```

Example 3.7.3

This example actually ships the remote computation call to the local site on the fly, and executes it locally.

1) go to JRelix running at "roo:~zwang26/JRelix/priv", invoke "compcall3" of "roo:~zwang26/public_aldatp/pubA", both of the "IN" and "OUT" parameter relations are located at "roo:~zwang26/JRelix/priv"

```
>aldatp://localhost/~zwang26/pubA/compcall3(in Stu6B, out R6);
> pr R6;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| Joe                  | 09/2001              |
| Sue                  | 09/2002              |
+----------------------+----------------------+
relation R6 has 2 tuples
>
```

Please pay attention to the difference between example 3.7.2 and example 3.7.3

3.8 Remote Statement Block

Remote statements enable us to execute statements remotely. To some extent, it works as if the users telnet to the remote system and operate that system.

Declaration Example

1) go to JRelix at "roo:~zwang26/public_aldatp"

2) declare domain

```
> pubA/{ domain authors string};
> pubA/{ domain Authors(authors)};
> pubA/{ let authors' be authors};
```

3) declare view

```
>pubA/{ V3 is Authors};
```

4) declare computation

```
> aldatp://localhost/pubA/{comp IntPerChg3(I,i,p) is
{I <- (1+i)**p-1}
alt
{i <- (1+I)**(1.0/p)-1}
alt
{p <- round(log(1+I)/log(1+i))}
};
```

Assignment Example

1)go to JRelix at "mimi:~zwang26/public_aldatp"

```
> aldatp://roo/~zwang26/pubA/{ Students3' <- Students3};
```

Update Example

1) go to JRelix at "roo: ~tim/public_aldatp"

```
> aldatp://roo/~zwang26/pubA/{update Students3 change name <- ''ccc'';};
```

Computation call Example

1) go to JRelix at "roo:~tim/public_aldatp"

```
> aldatp://localhost/~zwang26/pubA/{compcall3 (in Students3, out R3)};
```

Command Example

1) go to JRelix at "roo:~tim/public_aldatp"

```
>aldatp://localhost/pubA/{trace};
```

3.9 Remote Command

Compared with section 3.8.5, which demonstrates executing commands at the remote site, in this short section, we show examples of executing locally with a remote operand.

Example 3.9

1) go to "roo:~zwang26/public_aldatp", print remote relation

```
>pr pubA/Students3;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| aaa                  | 09/2001              |
| aaa                  | 09/2002              |
+----------------------+----------------------+
relation _temp_X9X_1 has 2 tuples
```

3.10 Start Options

In section 3.1, a demo environment and the essential way to launch multiple JRelix are presented. There are more options and situations will be elaborated in this section.

3.10.1 Root Level Server

If a root server is started on one physical machine, then by default all the JRelix systems running under a "public_aldatp" directory of that machine are publicly available. Otherwise, each JRelix instance is stand-alone, it can visit other publicly available JRelix, but it is not available to others.

Two kinds of root level server are implemented. There are a few differences between them.

The first one is called aldatpTLd, which means "aldatp top level daemon". As introduced in section 3.1, the way to start the aldatpTLd is:

The DBA or system administrator with the super user authority goes to the root user's "public_aldatp" directory (the "public_aldatp" at the upper right corner on Fig 3.1), and types "`java aldatpTLd &`".

Or

The aldatpTLd root server can also be launched by a normal user. The normal user goes to the "public_aldatp" directory, which should be immediately under his home directory, and type "`java aldatpTLd &`"

Example 3.10.1

```
[zwang26][roo][~/public_aldatp] java aldatpTLd &
[1] 96529
[zwang26][roo][~/public_aldatp] starting aldatp top level daemon, please wait
aldatp top level daemon is available now
```

The root server is responsible for listening to the "well-known" port number for aldatp, managing available port numbers, keeping a system dictionary, recording the information about all the sub server locations and their corresponding port numbers and responding to the requests from clients.

The aldatpTLd does not launch a JRelix instance. It is only a daemon program. By adding "java aldatpTLd" into one of the files under the /etc/rcX.d directory for Unix system, aldatpTLd root server could be started automatically when the Unix system is rebooted.

If the aldatpTLd is detected down, it can be started manually. The new started aldatpTLd can detect all the active running lower-level servers and do the house keeping work. A detailed description is given in Example 3.11.2

The second way to start a root level server is starting JRelix at "public_aldatp" directory while the aldatpTLd is not running. In this case, the JRelix not only takes the role of root server doing all the duties described above, (i.e. it starts a root daemon), but also starts a JRelix instance and acts as a normal JRelix server responding to the data processing requests from clients.

Example 3.10.2

1)If aldatpTLd is running, then stop it by using UNIX "ps" and "kill" commands to stop a process

```
[zwang26][roo][~/public_aldatp] ps
PID  TT  STAT      TIME COMMAND
......
96529  p1  I      0:00.00 /bin/sh /usr/local/bin/java aldatpTLd
96531  p1  S      0:00.26 /usr/local/jdk1.3.1/bin/i386/green_threads/java aldat
[zwang26][roo][~/public_aldatp/pubA] kill -9 96529 96531
```

2) launch JRelix at "public_aldatp"

```
[zwang26][roo][~/public_aldatp] java JRelix
starting top level aldatp server
+-----------------------------------------+
|            Relix Java version 0.80      |
| Copyright (c) 1997 -- 2002 Aldat Lab    |
|         School of Computer Science      |
|             McGill University           |
+-----------------------------------------+
>
```

If one starts the JRelix at the "public_aldatp" directory while the aldatpTLd is on, then a lower level JRelix server is launched. That means aldatpTLd still works as a root server, and the JRelix running at the "public_aldatp" only works as a normal lower-level server, which will be described in the next section.

Note : In this user manual, the aldatpTLd is launched at "public_aldatp" of a normal user "~zwang26", since the author has no root user authority. It's necessary to explain the difference between them

1) At "roo", if aldatpTLd is started at root user's "public_aldatp", then aldatp://roo/pubA/E8 and aldatp://roo/E9 refer to the E8 and E9 in Fig3.1. They are publicly available.

2)At "roo", if aldatpTLd is started at ~zwang26/public_aldatp, then aldatp://roo/pubA/E8 refers to "roo:~zwang26/public_aldatp/
pubA/E8" which does not exist. Consequently, E8 and E9 in Fig 3.1 are not publicly available. In this case, for example, aldatp://roo/~zwang26/pubA/E3 and aldatp://roo/pubA/E3 are the same thing.

Under normal situation, it is recommended to launch root server at root user's "public_aldatp" directory.

3.10.2 Lower-level JRelix Server

Each sub directory under "public_aldatp", is a residence of an individual JRelix instance. Even the directory ending with "public_aldatp" could reside an individual JRelix instance as well. By default, theses JRelix systems are publicly available. They are normal lower-level servers.

Example 3.10.3

1) Stop the JRelix launched in the previous example by typing "quit;" after ">"

```
>quit;
```

2) Start aldatpTLd,

```
[zwang26][roo][~/public_aldatp] java aldatpTLd &
```

3) Start JRelix "roo:~zwang26/public_aldatp".

```
[zwang26][roo][~/public_aldatp] java JRelix
Starting lower-level aldatp server using port:9993
...... //version info
```

3.10.3 Protected JRelix Server

Besides those directories under "public_aldatp", JRelix could also be launched at any directory outside "public_aldatp". In this case, by default the resource of the JRelix instance are public unavailable to any other users on the Internet as well as any other users on the local machine. The only exception to this case is "remote view", which will be illustrated in Section 3.13.2.

Example 3.10.8

1) Make sure aldatpTLd has already been started.

2) Go to "roo:~zwang26/jrelix/priv" and start JRelix

```
[zwang26][roo][~/jrelix/priv] java JRelix
Starting protected lower-level aldatp server using port:9991
...... //version info
```

3.10.4 Stand-alone JRelix

Under some circumstances, we would prefer starting a stand-alone JRelix to starting a JRelix server. For example, when the owner is rebuilding or reconstructing a database under "public_aldatp", or the owner wants to possess the database exclusively for some reason even if the database is under "public_aldatp" directory. The owner can launch a database in stand-alone mode so that it is absolutely prevented from being accessed from outside.

Example 3.10.9

```
[zwang26][roo][~/public_aldatp] java JRelix -SA
Starting stand-alone JRelix.
...... //version info
```

Note: 1) It is not good to start more than one copy of JRelix concurrently at the same directory, since the current JRelix is RAM based and concurrency control is not implemented yet.

2) A "stand-alone" JRelix does not mean it is isolated from the outside. A stand-alone JRelix still has the freedom to access any outside database only if that database is publicly available. In other words, launching a stand-alone JRelix just means that it can not be accessed from outside but it can access outside data.

3) Stand-alone jrelix could be launched under "public_aldatp" or outside "public_aldatp".

Under some unusual circumstances, we intend to start a JRelix server but the system automatically starts a stand-alone JRelix. One possibility is the top level server is not running or the communication to the top level server fails. Consequently, it is forced to start a stand-alone JRelix. Another unusual situation is that all the reserved port numbers are consumed, thus no more port numbers are available. In this case, it is also forced to start a stand-alone JRelix. More will be explained in section 3.11 .

Example 3.10.10

```
[zwang26][roo][~/public_aldatp/pubA] java JRelix
Couldn't get I/O for the connection to TOP level server
Starting stand-alone JRelix.
...... //version info
```

3.11 Manage Port Numbers

On each physical machine, there is only one "well-known" port for the aldatp. This "well-known" port is used by the root server.

As it can be seen, each lower-level server also consumes one port number. These lower-level servers start or shut down dynamically, thus they consume the ports dynamically. At one time, usually only small parts of this big multiple database family are active. In addition, the multiple database system could contain huge numbers of members. Furthermore, the multiple database system is not static itself. Making new databases, moving databases, or removing databases are permitted. In short, the active status of the entire multiple system may keep changing dynamically and frequently. So it is unwise to make all port numbers for each member in the multiple system to be publicly "well-known". One solution is selecting a master on each physical computer to manage these ports and these ports are totally transparent to end-users. In our implementation, the root server is such a master.

Suppose we estimate that the maximum number of members in the multiple database system is N. First we select M available port numbers. Suppose on average, one third of the N members are active, then we could choose M to be equal to or greater than N /3. A little bit more is better. Usually this job is done by DBA or System Administrator, who manually edits a ".ports" file at the "public_aldatp" directory. In this file, at each line we write down one of these M port numbers. After that, we could start the aldatpTLd root server. Lower-level servers could also be launched afterwards. The lower-level server would ask the root server to allocate an available port for it. The root server receives the request, then picks one of the available ports from these M ports and allocates it to the requester.

It is possible that the real time number of active members is greater than M. It is also possible that some of the M ports are consumed by other applications. Thus, when the root server is asked for allocating a port, no more port is available. If this happens, the system would complain "no available port". One solution is to edit the ".ports" file and add more available port numbers. The root server and all the active lower-level servers need not be restarted. Another alternative solution is to wait until one of the M ports is released, then it can be reused for new requester.

Example 3.11.1

1) Quit all the JRelix and aldatpTLd started in the previous examples.

2) Edit ".ports" file at "roo:~zwang26/public_aldatp" so that it only contains three port numbers.

```
9991
9992
9993
```

3) Start a program that consumes one of these three ports

```
[zwang26][roo][~] java HTTPserver 9991
```

4) Go to "roo:~zwang26/public_aldatp", start aldatpTLd

```
[zwang26][roo][~/public_aldatp] java aldatpTLd &
```

5) Start a program that consumes another port

```
[zwang26][roo][~] java HTTPserver 9992
```

6) Go to "roo:~zwang26/jrelix/priv", start JRelix

```
[zwang26][roo][~/jrelix/priv] java JRelix
Starting protected lower-level aldatp server using port:9993
...... //version info
```

7) Go to "roo:~zwang26/public_aldatp", start JRelix

```
[zwang26][roo][~/public_aldatp] java JRelix
Warning! NO available port
Starting stand-alone JRelix.
...... //version info
```

8) Quit the JRelix started at step 7); Edit ".ports" file at "roo:~zwang26/public_aldatp", and add one more port 9994 at the bottom of the file

9) Go to "roo:~zwang26/public_aldatp", start JRelix

```
[zwang26][roo][~/public_aldatp] java JRelix
Starting lower-level aldatp server using port:9994
...... //version info
```

Up till now, there are totally 4 ports in the ".servies" file. 9991 and 9992 are used by two programs named "HTTPserver", 9993 and 9994 are consumed by two JRelix running at "roo:~zwang26/jrelix/priv" and "roo:~zwang26/public_aldatp" respectively.

10) Stop the HTTPserver using port 9991, so the 9991 is released.

11) Go to "roo:~zwang26/public_aldatp/pubA", start JRelix. The new released 9991 port will be consumed .

```
[zwang26][roo][~/public_aldatp/pubA] java JRelix
Starting lower-level aldatp server using port:9991
...... //version info
```

12) Stop the HTTPserver using port 9992

13) Go to "roo:~zwang26/public_aldatp/pubA/pubA1", start JRelix

```
[zwang26][roo][~/public_aldatp/pubA/pubA1] java JRelix
Starting lower-level aldatp server using port:9992
...... //version info
```

If the root server is shut down while some lower-level server is still running. Afterwards, when the root server is restarted, the root server can detect all the current alive lower-level servers.

Example 3.11.2
After example 3.11.1, four lower-level servers are running,

```
9991: pubA
9992: pubA1
9993: priv
9994: public_aldatp
```

1) Stop aldatpTLd by using UNIX "ps" and "kill" command to stop a process
2) Go to JRelix on "roo:~zwang26/public_aldatp/pubA/pubA1", try to print the relation named Students3" on "roo:~zwang26/public_aldatp/pubA"

```
>pr aldatp://localhost/~zwang26/pubA/Students3;
>java.net.ConnectException: Connection refused
>InterpretError: service temporarily unavailable
>
```

Note: Since the root server is down, the remote request is declined.

3) Quit JRelix at "roo:~zwang26/jrelix/priv", so there are three alive JRelix lower-level servers now

```
9991: pubA
9992: pubA1
9994: public\verb_aldatp
```

4) Go to "roo:~zwang26/public_aldatp", start aldatpTLd

```
[zwang26][roo][~/public_aldatp] java aldatpTLd &
```

Note: the root server can detect the three alive lower-level servers and their port numbers

5) Go to JRelix on "pubA1", try to print the relation named "Students3" on "roo:~zwang26/public_aldatp/pubA" once more

```
>pr aldatp://localhost/~zwang26/pubA/Students3;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| aaa                  | 09/2001              |
| aaa                  | 09/2002              |
+----------------------+----------------------+
```

Now there are three alive JRelix lower-level servers

```
9991: pubA
9992: pubA1
9994: public_aldatp
```

7) Go to "roo:~zwang26/public_aldatp/pubB", start JRelix

```
[zwang26][roo][~/public_aldatp/dirB] java JRelix
Starting lower-level aldatp server using port:9993
...... //version info
>
```

3.12 Background Server

It is not mandatory to start the lower-level servers manually. Instead, the root server can start lower-level servers automatically if necessary.

Example 3.12.1

1) Quit all the alive lower-level servers and aldatpTLd if applicable.

2) Start aldatpTLd

```
[zwang26][roo][~/public_aldatp] java aldatpTLd &
```

3) Go to "roo:~zwang26/jrelix/priv", start JRelix

```
[zwang26][roo][~/jrelix/priv] java JRelix
Starting protected lower-level aldatp server using port:9993
...... //version info
```

4) Now only aldatpTLd and JRelix at "roo:~zwang26/jrelix/priv" is running

```
[zwang26][roo][~] ps
  PID  TT  STAT      TIME COMMAND
74405  p0  I+     0:00.00 /bin/sh /usr/local/bin/java aldatpTLd
74407  p0  I+     0:00.26 /usr/local/jdk1.3.10/bin/i386/green_threads/java aldatpTLd
74438  p2  I+     0:00.00 /bin/sh /usr/local/bin/java JRelix
74440  p2  I+     0:00.50 /usr/local/jdk1.3.10/bin/i386/green_threads/java JRelix
......
```

5) Go to JRelix at "roo:~zwang26/jrelix/priv", try to visit JRrelix at "roo:~zwang26/public_aldatp/pubA"

```
>pr aldatp://localhost/~zwang26/pubA/Students3;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| aaa                  | 09/2001              |
| aaa                  | 09/2002              |
+----------------------+----------------------+
```

A background JRelix server is started automatically at "roo:~zwang26/public_aldatp/pubA" by the root server.

6) Check the process again.

```
[zwang26][roo][~] ps
  PID  TT  STAT      TIME COMMAND
74405  p0  I+     0:00.00 /bin/sh /usr/local/bin/java aldatpTLd
74407  p0  I+     0:00.30 /usr/local/jdk1.3.10/bin/i386/green_threads/java aldatpTLd
74577  p0  I+     0:00.00 /bin/sh /usr/local/bin/java JRelixBack
74579  p0  I+     0:00.45 /usr/local/jdk1.3.10/bin/i386/green_threads/java JRelixBack
74543  p2  I+     0:00.00 /bin/sh /usr/local/bin/java JRelix
74545  p2  I+     0:00.52 /usr/local/jdk1.3.10/bin/i386/green_threads/java JRelix
......
```

7) Go to "roo:~zwang26/public_aldatp/pubA", try to start a JRelix server

```
[zwang26][roo][~/public_aldatp/pubA] java JRelix
Warning! JRelix server is running on this directory
It is forbidden to start more than one JRelix server concurrently at the same directory.
```

8) Go to "roo:~zwang26/public_aldatp" start a JRelix and visit "roo:~zwang26/public_aldatp/pubA"

```
[zwang26][roo][~/public_aldatp] java JRelix
Starting lower-level aldatp server using port:9992
...... //version info
>pr pubA/Students3;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| aaa                  | 09/2001              |
| aaa                  | 09/2002              |
+----------------------+----------------------+
```

A JRelix background server is available to any other JRelix.

9) Quit the JRelix at "roo:~zwang26/jrelix/priv"

```
>quit;
 waiting all the active threads finish working ...
[zwang26][roo][~/jrelix/priv]
```

The background server running at "roo:~zwang26/public_aldatp/pubA" will be terminated as well, since "roo:~zwang26/public_aldatp/pubA" is launched on account of "roo:~zwang26/jrelix/priv". Although after launching, "roo:~zwang26/public_aldatp/pubA" could be used by anyone else.

10) Check process again

```
[zwang26][roo][~] ps
  PID  TT  STAT      TIME COMMAND
74405  p0  I+     0:00.00 /bin/sh /usr/local/bin/java aldatpTLd
74407  p0  I+     0:00.32 /usr/local/jdk1.3.10/bin/i386/green_threads/java aldatpTLd
74646  p3  I+     0:00.00 /bin/sh /usr/local/bin/java JRelix
74648  p3  I+     0:00.54 /usr/local/jdk1.3.10/bin/i386/green_threads/java JRelix
......
```

Only aldatpTLd and JRelix at "roo:~zwang26/public_aldatp" are running. If "roo:~zwang26/public_aldatp" visit "roo:~zwang26/public_aldatp/pubA" again, a new background server at "roo:~zwang26/public_aldatp/pubA" will be launched.

3.13 Security Issue

3.13.1 Protected Server

As presented in section 3.2, the JRelix systems outside "public_aldatp" are private and protected, since there are no ways to parse a directory outside "public_aldatp". By default, only the owners can operate these systems at local sites. However, if a user defines a remote view which refers to local resource, it is desirable to permit remote sites to access the local resource when the view is being actualized.

Example 3.13.1

1) At "roo:~zwang26/jrelix/priv", we define a view on "mimi:~zwang26/public_aldatp/pubA" , and this view refers to the relation named "Stu6B" at "roo:~zwang26/jrelix/priv".

```
>aldatp://mimi/~zwang26/pubA/V3 is Stu6B;
OK
>
```

When this view is actually used, the remote database running on "mimi:~zwang26/ public_aldatp/pubA" would access data from "roo:~zwang26/jrelix/priv" to actualize the view. So the "roo:~zwang26/jrelix/priv" must possess the sever ability to respond to this data access request. But we do not want it to be publicly available. This is the reason why it is outside the "public_aldatp" directory.

2) Go to JRelix at "mimi:~zwang26/public_aldatp/pubA" to actualize the view

```
>R3 <- V3;
>pr R;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| Joe                  | 09/2001              |
| Sue                  | 09/2002              |
+----------------------+----------------------+
relation R has 2 tuples
>
```

3) Stop JRelix at "roo:~zwang26/jrelix/priv"

4) Go to JRelix at "mimi:pubA" to check the view once more

```
>pr V3;
+---------------------+---------------------+
| name                | date                |
+---------------------+---------------------+
| Joe                 | 09/2001             |
| Sue                 | 09/2002             |
+---------------------+---------------------+
relation _temp_X9X_14 has 2 tuples
>
```

A background JRelix server is started at "roo:~zwang26/jrelix/priv"

3.13.2 File Access Permission

In general, the JRelix under "public_aldatp" are publicly available, while the JRelix outside "public_aldatp" are private and protected. By this way, we control the access permission at a database or directory level. However, we are able to control the access permission more finely at a relation or file level.

Example 3.13.2

1) Start aldatpTLd, start JRelix at "roo:~zwang26/public_aldatp/pubA";

2) In this example, Students3 is a relation of "roo:~zwang26/public_aldatp/pubA". Go to "roo:~zwang26/public_aldatp/pubA", change the mode of Students3 to be only readable and writable by the owner user.

```
[zwang26][roo][~/public_aldatp/pubA] chmod 600 Students3
```

3) Go to "roo:~zwang26/jrelix/priv", start JRelix, and try to read Students3 at "roo:~zwang26/public_aldatp/pubA"

```
[zwang26][roo][~/jrelix/priv] java JRelix
Starting protected lower-level aldatp server using port:9993
...... //version info
>pr aldatp://localhost/~zwang26/pubA/Students3;
InterpretError: Permission denied to read Students3
```

4) Go to "roo:~zwang26/public_aldatp/pubA", change the mode of R to be readable by all but only writeable by owner

```
[zwang26][roo][~/public_aldatp/pubA] chmod 644 Students3
```

5) Go to "roo:~zwang26/jrelix/priv", try to read Students3 of "roo:~zwang26/public_aldatp/pubA" again

```
>pr aldatp://localhost/~zwang26/pubA/Students3;
+----------------------+----------------------+
| name                 | date                 |
+----------------------+----------------------+
| aaa                  | 09/2001              |
| aaa                  | 09/2002              |
+----------------------+----------------------+
relation _temp_X9X_1 has 2 tuples
>
```

6) At "roo:~zwang26/jrelix/priv" try to update Students3 of "roo:~zwang26/public_aldatp/pubA"

```
>update aldatp://localhost/~zwang26/pubA/Students3 change name <- "Joe";
Permission denied to write Students3
```

7) Go to "roo:~zwang26/public_aldatp/pubA", change the mode of Students3 to be readable and writable by all

```
[zwang26][roo][~/public_aldatp/pubA] chmod 666 Students3
```

8) Go to JRelix at "roo:~zwang26/jrelix/priv" try to update Students3 of "roo:~zwang26/public_aldatp/pubA" again

```
>update aldatp://localhost/~zwang26/pubA/Students3 change name <- "Joe";
OK
```

Chapter 4

Implementation of JRelix Distributed Systems

In this chapter, we are going to describe the implementation of JRelix distributed systems in detail. In section 4.1, we will give an overview of the current JRelix system architecture. Section 4.2 presents the general implementation issues on aldatp. In section 4.3, more detail and specific implementation issues for different remote functions of JRelix are illustrated, such as the capability of executing assignments, updates, commands, declarations, and invoking computation calls across the Internet. The implementation of system administration for aldatp is described in section 4.4. The system administration topics cover starting the system, managing port numbers, launching background servers and security issues on data access permission.

4.1 JRelix System Overview

Architecture

The JRelix system contains four main parts, the Parser, the Interpreter, the Execution Engine and the Data. The parser and the interpreter function as the front-end processor and act as an interface between end-user and the central execution engine. The central execution engine fulfills the tasks passed from the interpreter. Data are permanently stored files as well as run-time data in RAM. They consist of not only the user-defined relational data and stored computation code, but also "system tables" which represent the system information of the database. Figure 4.1 is an overview of the system. The underlying communication protocol is TCP/IP socket.

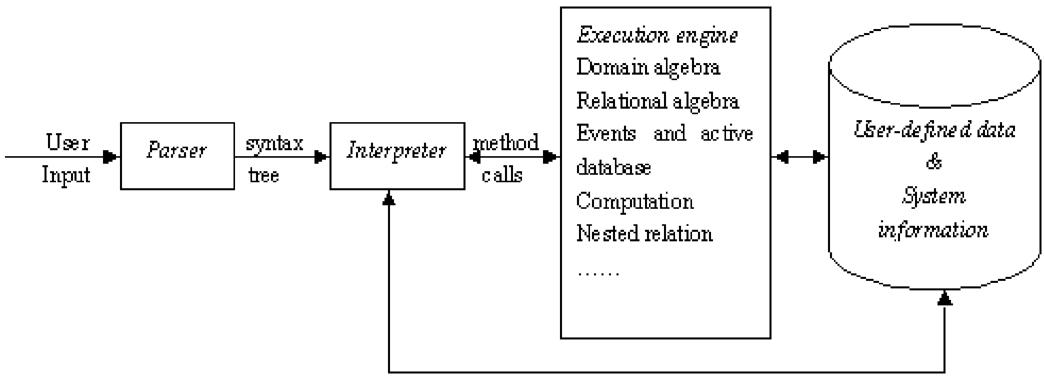


Figure 4.1: JRelix System

Parser

A JRelix command entered by end-user is first accepted by JRelix parser. The parser reads the command-line input, analyzes the command syntax and finally translates the command into an intermediate code which has a tree structure and is therefore called syntax tree. The parser is created by using Java Compiler Compiler (JavaCC) [SDV96] , a parser generator that reads a high-level grammar specification and converts it to a Java Program that can recognize matches to the grammar. JJTree is a preprocessor for JavaCC that builds parser trees. The output of JJTree is run through JavaCC to create the parser. In JRelix implementation, a Parser class is created corresponding to this module.

Interpreter

The interpreter repeatedly calls the parser, receives trees passed from the parser, traverses the syntax tree and decomposes it into a set of method calls executed by the execution engine. The interpreter also interacts with the system tables to retrieve and update information about attributes, relations, views and computations in the database. In JRelix implementation, an Interpreter class is built to represent the interpreter.

Execution Engine

Essentially, there are five conceptual aspects in JRelix system: relational algebra, domain algebra, computation, events and nested relation. They corresponds to five basic function modules, i.e. Relation Processor[Hao98], Virtual Domain Actualizer[Yuan98], Computation Processor[Bak98], Events and Active Database[He97], and Nested Relation Processor[Hao98], which work together and also support each other to fulfill the tasks of the execution engine. Apart from the core function modules, there are other function modules such as Geditor[Che01] and Attribute Meta data[Mer01] etc.

Memory and Disk Files

The data of each User-defined relation is permanently stored in a disk file having the same name as the relation and read fully into RAM when referred (assume they are small enough).

Apart from user-defined relation data, JRelix maintains so-called “system information” which contains important information about user-defined relations and domains in the system, describes the current system execution state and controls system behavior either during a single JRelix session or across multiple sessions. The system relations .rel and .dom store information about all the relations (including views and computations) and attributes in the database respectively, while system relation .rd stores information that links the relations with attributes defined on. File .expr contains the serialized syntax tree of views and virtual attributes, while .comp contains the serialized syntax trees of computations. System tables exist both in the memory and on the hard disk with different formats.

4.2 General Issues on JRelix Distributed Data Processing Implementation

As illustrated in chapter 1, the basic idea of ALDATP is adapting a URL-based name extension to a database programming language "aldat", which gives it collaborative and distributed capability over the Internet with no changes in syntax or semantics apart from the new structure in names. New components are added to the current system and some existing models are modified to implement the multiple collaborative system. For instance, the parser is modified to accept URL-based name structure, the interpreter is upgraded to recognize and analyze the syntax tree with remote query. In the execution engine, new components for aldatp are implemented to deal with distributed data processing.

4.2.1 Parsing Aldatp Syntax and Building Syntax Tree

Syntax specification

The syntax for aldatp has already been introduced in the Section 3.2. A demon environment as shown in Figure 3.1 and Table 3.1 gives thoroughly new syntax examples of the URL-based name structure for that sample multiple database trees. Here we summarize it as follows:

aldatpheader element

The element could be one of the followings:

1) identifier of a relation, view or computation

2) parenthesized expression

3) statement or statement block in a brace

The following is the specification for ”aldatpheader”.

```
TOKEN :                 /* ALDATPHEADER */
{
        < ALDATPHEADER : <FULL_ALDATPHEADER> | <SHORT_ALDATPHEADER> >
        |
        < FULL_ALDATPHEADER : <FULL_ALDATP> ((<LETTER>)+ (<DOT>)*)+
         <SLASH> ( <TILDE> <IDENT> <SLASH>)*  (<IDENT> <SLASH>)*  >
        |
        < SHORT_ALDATPHEADER : <IDENT> (<SLASH> <IDENT>)* <SLASH>  >
        |
        < FULL_ALDATP : "aldatp://" >
        |
        < IDENT : <LETTER> (<LETTER>|<DIGIT>|<OTHERS>)* >
   |
   < #SLASH : ["/","\\"] >      // ‘‘/’’ for Unix and ‘‘\\’’ for Windows
        |
        < #TILDE : [ "~" ] >
        |
   < #LETTER : ["a"-"z", "A"-"Z"] >
        |
        < #OTHERS : ["_", "’"] >
|
< DOT : "." >
}
Note
e1 | e2 | e3 | ... : A choice of e1, e2, e3, etc.
( e )+             : One or more occurrences of e
( e )*             : Zero or more occurrences of e
["a"-"z"] matches all lower case letters
```

There are two kinds of ALDATPHEADER, one is full format and the other is short format. The full format is used under most situations. Its structure is as follows:

1) First begins with “aldatp://”, which is a header,

2) Then follows by the host name `((<LETTER>)+ (<DOT>)*)+ <SLASH>`, e.g. “roo/” or “mimi.cs.mcgill.ca/”

3) Next follows by an option part `( <TILED> <IDENT> <SLASH>)*`. It specifies the alias of user home directory, e.g.“~zwang26/”,

4) The last part `(<IDENT> <SLASH>)*` is optional. If the destination path ends with “public_aldatp”, then this part does not occur. Otherwise it is the rear part of the target path, which is immediately after “public_aldatp”.

The short format is much simpler. But it only applies to parent JRelix accessing its descendants. The syntax is `<IDENT> (<SLASH> <IDENT>)* <SLASH>`, e.g. “pubA/pubA1/”

To implement the above syntax, we first modify the grammar specification text file, Parser.jjt, by adding the specification of aldatp syntax. Then we generate the new Parser using JJTree and JavaCC.

4.2.2 Building the Parser Tree

The syntax tree generated by the parser is compact and easy to interpret. Figure 4.2 shows the syntax tree for assignment : `A <- R ijoin S`

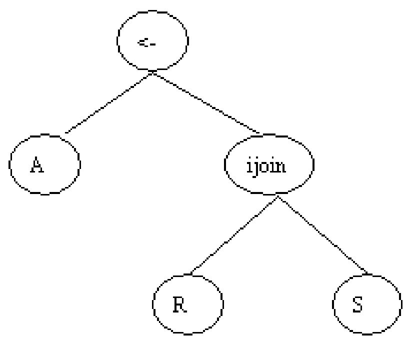


Figure 4.2: Syntax tree for assignment

In Figure 4.2, if the element is an identifier, e.g. relation R, it corresponds to a single leaf node in the tree. If the element is an expression, e.g. R ijoin S, then it corresponds to the sub-tree rooted at node "ijoin". Last of all, if the element is a statement, e.g. `A <- R ijoin S`, then it corresponds to the whole syntax tree.

Each node in the syntax tree contains fields indicating the nature of the node. The principle fields are described in the Table 4.1

There are in total more than two hundreds different types and opcodes (operation code) defined in JRelix system. To save space, we list some of them that will be used in this paper in table 4.2.

We can dump the parser tree to a flat text file according to some traverse order. With this dump file and traverse order, the parser tree can be recreated. The following is the dump file for syntax tree of `A <- R ijoin S`. Each line corresponds to one node of the tree

item	**type**	**description**
parent	Node	Parent node
children	java.util.Vector	Children nodes
type	int	Operation type, e.g. declaration, update
opcode	int	Specific operation, e.g. declare relation, declare view, update add, update change
name	String	The name of the identifier, If the node is an identifier
info	Object	User-input data for some sort of nodes
identifier	String	Node identifier

Table 4.1: The principle fields of syntax tree nodes

type	**opcode**
100 statement	101 sequence ...... 110 execute 111 statement block
140 declaration 1140 left-hand view declaration	141 relation 142 view 143 domain 144 let 145 computation
160 assignment 1160 left-hand assignment	161 Assignment 162 Incremental assignment
180 update 1180 left-hand update	181 update add 182 update delete 183 update change
230 identifier	230 identifier
301 binary join operator	361 ijoin 362 ujoin 363 sjoin 364 ljoin 365 rjoin ......
......	......

Table 4.2: Sample operation types and codes in JRelix system

```
identifer:Assignment name:null opcode:161 type:160 numChildren:2 info:null url:null
identifer:Identifier name:A opcode:230 type:230 numChildren:0 info:null url:null
identifer:Join name:null opcode:361 type:301 numChildren:2 info:null url:null
identifer:Identifier name:R opcode:230 type:230 numChildren:0 info:null url:null
identifer:Identifier name:S opcode:230 type:230 numChildren:0 info:null url:null
```

In our URL-based name solution, we can identify each database element on the Internet. In order to obtain such capability, we modify the previous node structure by adding a new field url. This field contains the URL information of the node.

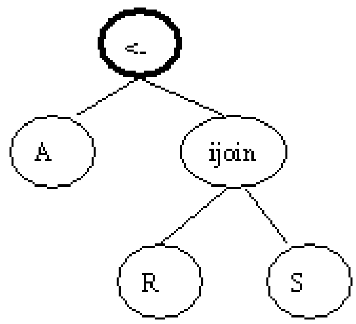


Figure 4.3: Syntax tree for a remote assignment

identifier	name	opcode	type	Numchildren	info	url
StatementBlock	null	111	100	1	null	aldatp://roo/pubA/
Assignment	null	161	160	2	null	null
Identifier	A	230	230	0	null	null
Join	null	361	301	2	null	null
Identifier	R	230	230	0	null	null
Identifier	S	230	230	0	null	null

Table 4.3: Dumped syntax tree for a remote assignment

For the assignment `A <- R iojin S`, there are many extensions when remote data processing capability is available. Figure 4.3 and Table 4.3 give the syntax tree and its dumped file record for `aldatp://roo/pubA/{A <- R ijoin S}`; Figure 4.4 lists more extensions. To save space, we only give the syntax tree and indicate those nodes having a URL that is not null.

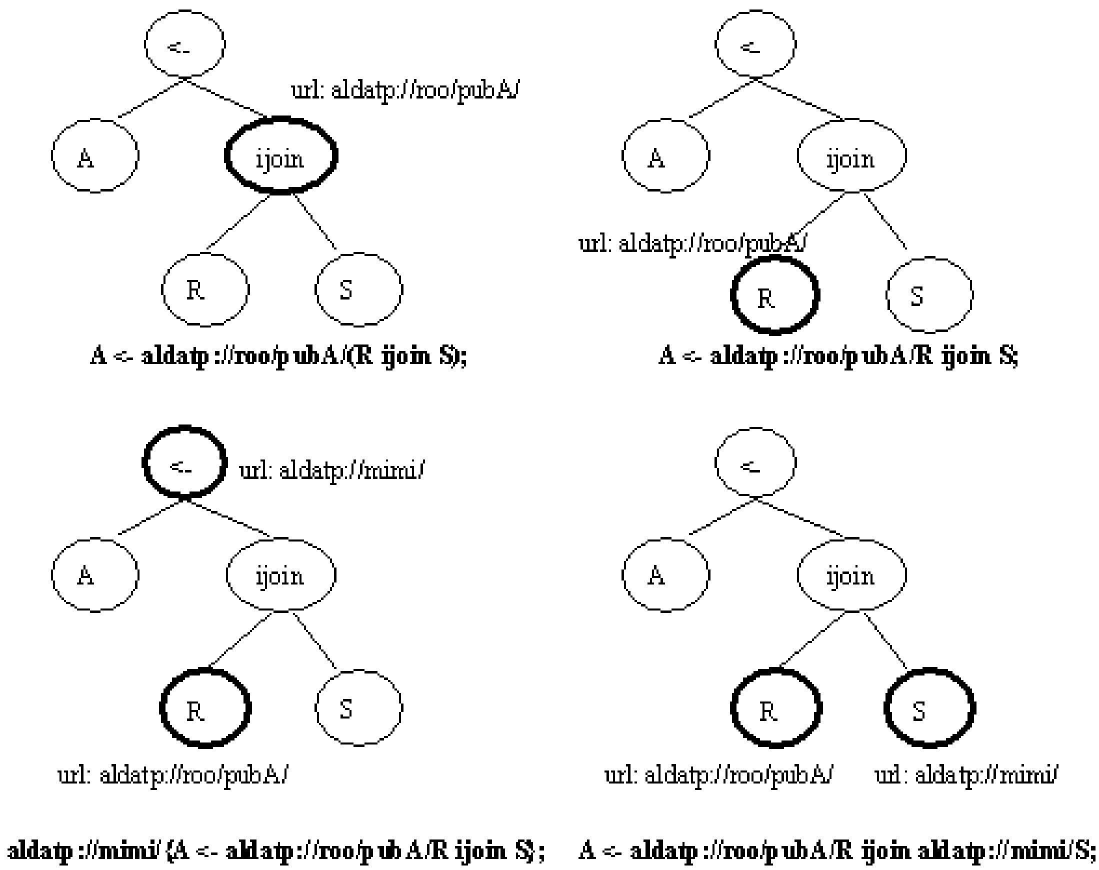


Figure 4.4: Examples of non-null URLs in a syntax tree

4.2.3 Overall Process Flow

So far, the parser tree with URL information is generated. Now we illustrate how the syntax tree with non-null URL nodes is processed in general. Figure 4.5 shows the system architecture of JRelix with distributed capability.

Interpreter

A JRelix command entered by an end-user is first accepted by JRelix parser and translated into a syntax tree. While the interpreter is traversing and decomposing the tree into a set of method calls executed by the execution engine, if it encounters a

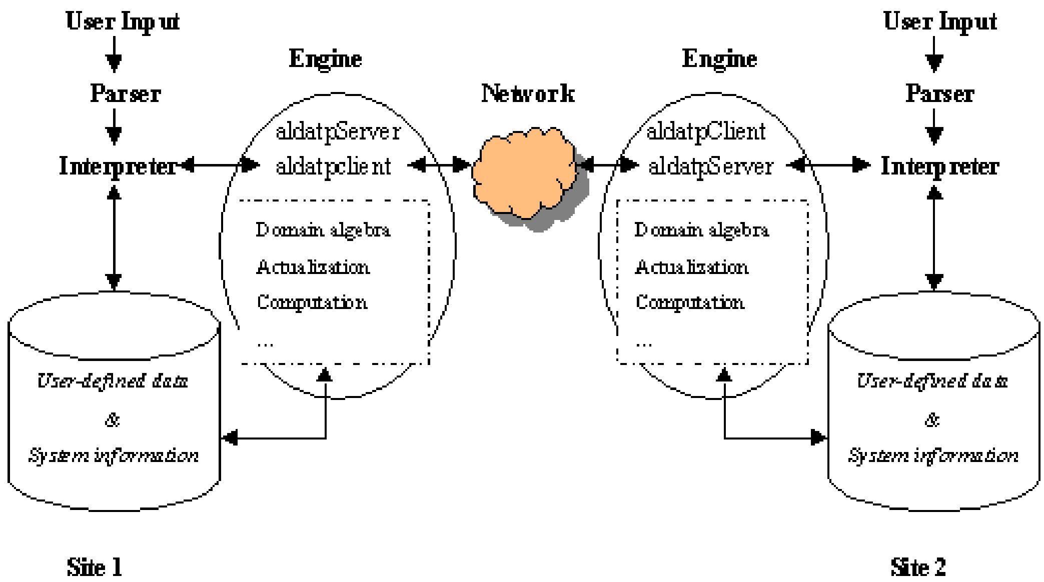


Figure 4.5: JRelix Multidatabase System Architecture

tree node with non-null URL, it calls "aldatpClient" to process the sub-tree rooted at this node and then waiting for the response from the "aldatpClient". As we know, the sub-tree might be as large as the whole syntax tree or as small as a leaf node of the whole syntax tree. The types of the sub-trees are various. It is possibly a relational expression, thus the expected result is a relation. It is probably a statement or command, so the expected result is a response code about whether the statement or command is executed successfully. After the Interpreter receiving the result of remote processing from "aldatpClient", it continues the unfinished work. It's possible that there are several non-null nodes in a syntax tree, thus the interpreter will call the aldatpClient more than once.

Note

If the sub-tree is a single leaf node indicating a remote relation for example, when the node occurs at the right hand of the statement, a corresponding relation is fetched from the target site. In fact, it can be considered as a special case of expression. However, if the node occurs at the left hand of the statement, e.g.

```
aldatp://mimi/A <- R ijoin S,
aldatp://mimi/V is R;
update aldatp://mimi/A add R;
compcall ( in R, out aldatp://mimi/A);
```

there are different ways for different situations that will be discussed in Section 4.3. In this section, we only discuss the right-hand case to make it easy at the beginning.

aldatpClient

When an "aldatpClient" receives a request from the Interpreter, it extracts the URL from the root node of the sub-tree being processed. From analyzing the URL, it gets the host name and path information of the target JRelix. Then it connects with the root server on that host, enquires and gets the port number of the server started at the target path. With the host name and port number, the aldatpClient can build a direct connection to the target aldatpServer and communicate with that server. After the connection is established, the "aldatpClient" dumps the sub parser tree in a flat text format as shown in 4.2.1 and send it to the target server. Then the "aldatpClient" blocks and waits for the response from the target server. The response from the target server is either a response code or a response code plus a relation. For the latter, the "aldatpClient" will build a corresponding relation in the local environment.

aldatpServer

An aldatpServer can be started in different ways. In addition, there are two distinct types of aldatpServer. One is regular and the other is protected. When an aldatpServer is started, it continuously listens to the port, accepts connections from aldatpClients, and creates new threads aldatpHandler to handle the client requirements. If the request from the client is a parser tree in a flat text format, the aldatpHandler will create the parser tree first. Then it calls the local interpreter to interpret the tree. Afterwards, it sends the result back to the caller.

Figure 4.6 explains the relationship between aldatpClient, aldatpTLd, aldatpServer and aldatpHandlers.

1) Interpreter encounters a URL node and calls aldatpClient.

2) aldatpClient gets the host name from the URL and uses the "well-known" port to connect with the root server to ask for the port of the target JRelix ("aldatpServer" in the diagram).

3) The root server supplies the port of the target JRelix server to the client.

4) aldatpClient uses the port from step 3) to connect with the target JRelix server(aldatpServer).

5) aldatpServer creates new threads to handle the client request.

6) If the request is "ParserTree" then calls local Interpreter to deal with it.

7) Interpreter calls Execution Engine

8) - 11) The result is returned from the Execution Engine by the Interpreter and the aldatpHandler to the aldatpClient.

4.2.4 Shipping Query and Shipping Data

Except for some administration messages, the typical things sent across the network are queries and data. The most common model is the client sends a parser tree to the server, consequently the server sends back a response code or a response code plus a relation.

In the term of aldatp, a query is a statement, command or relational expression. A statement is one of the assignments, updates, declarations, loop statements, conditional statements, etc. In the point of view of a syntax tree, a query is a sub-tree or the whole tree. Shipping query is explained in detail in Section 4.2.2. In summary, the way to ship query is to first dump the syntax tree into a flat text format stream and then transfer it to the target site. The receiver gets the text format records and then recreates the parser tree.

Shipping data is more complicated. The data we mentioned here is the broad sense data. It not only refers to user-defined relational data (flat or nested) and attribute

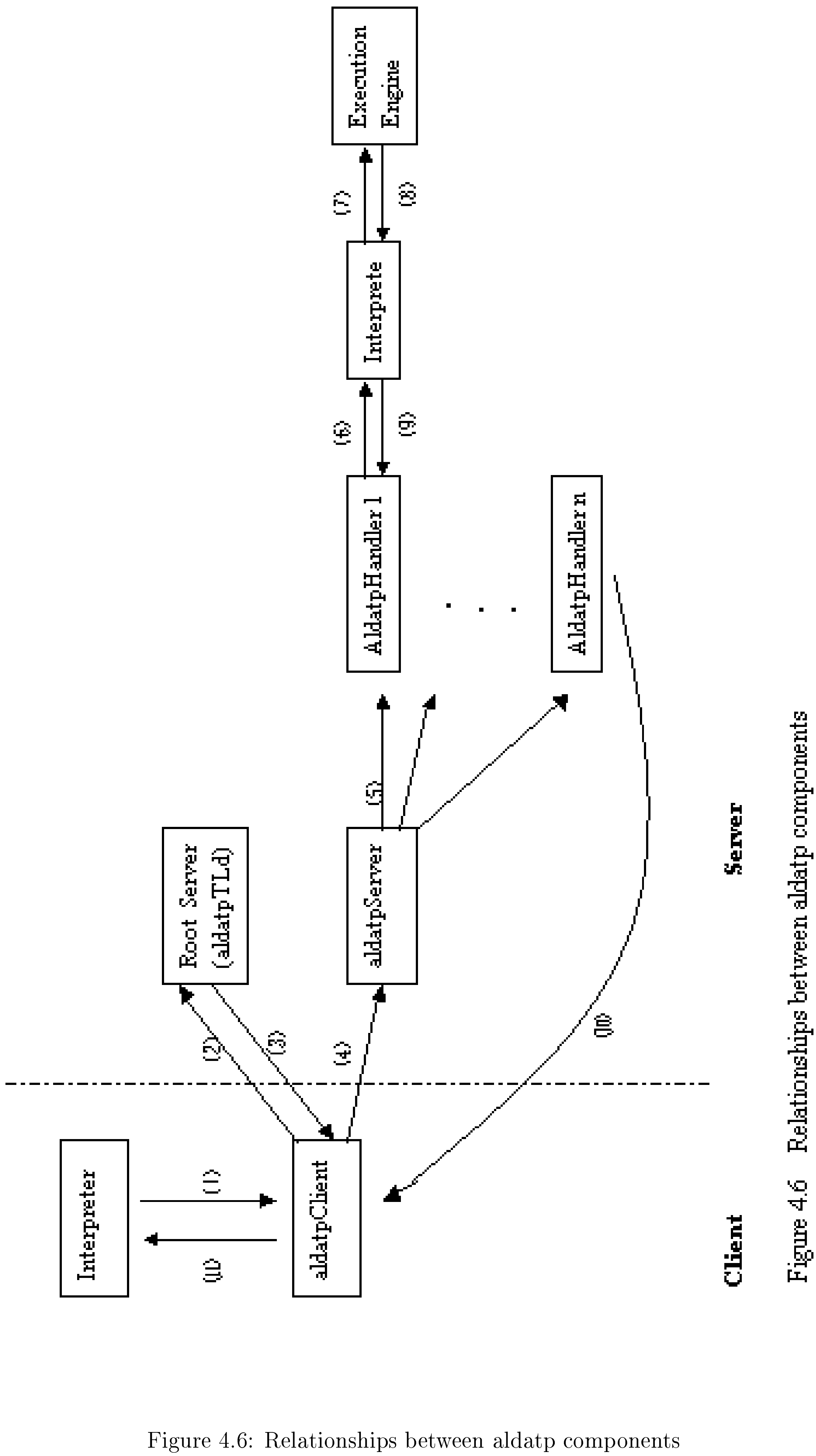


Figure 4.6: Relationships between aldatp components

information; it also includes the "code" or syntax tree for a computation. Before introducing shipping data, we will explain how the data is organized and stored in the JRelix system first.

Upon declaration and initialization, a relation is stored in a file whose name corresponds to the name of the relation. Every JRelix system maintains a set of "system tables" which represent the data dictionary of the database. They are stored in permanent files on the hard disk as well as memory in RAM. Once the interpreter starts, two system tables DomTable and RelTable are constructed in RAM from loading files .rel, .dom, .rd, .comp and .expr on the disk.

System relation	attributes	descriptions
.rel	.rel_name	Relation name
	.tuples	Number of tuples
	.attributes	Number of attributes
	.rvc	Type (relation. view, computation)
	.sort	Number of sorted attributes
.dom	.dom_name	Attribute name
	.type	Attribute type
	.count	Number of times this attribute is referenced
.rd	.rel_name	Relation name
	.dom_name	Attribute name
	.position	Position of this attribute

Table 4.4: Definition of system relations: .rel, .dom, and .rd

Shipping flat relation

Figure 4.7 is an example relation R, the relative information about R in the system relations .dom, .rel, .rd, and its corresponding data file stored on the hard disk.

To ship data, not only the data file whose name corresponds to the name of the relation is shipped, but also the relational and domain information stored in the "system tables" are shipped. To do so, we should pay attention to the following three points.

First, the JRelix is RAM based. In other words, .dom, .rel and .rd files on the disk are not updated immediately. However, the JRelix run time system updates

```
  R ( name    date   )
      Sam    690810
      Joe    661120

.dom          .rel             .rd            R
name:7:1      R:2:2:15:2       R:name:0       Sam : 690810
date:7:1      ...              R:date:1       Joe : 661120
...                            ...
```

Figure 4.7: Example of system relation and user defined relation

the system tables DomTable and RelTable in real time. Thus, to transfer system information, we can't rely on the .dom, .rel and .rd files. Instead, we extract the information from the system tables and transfer them. At the receiver site, the relational and domain information about the shipped data are written into the system tables and run time environment in RAM. When the JRelix session is finished, the system dictionary will be written back into disk files .rel, .dom, .rd, .comp and .expr.

Second, when the data file and system information is received, a new relation is created accordingly. For the "right hand" situation, the newly created relation is temporary. For example,

```
A <- aldatp://mimi/R ijoin S.
```

R is shipped from "mimi" to local database. It is not necessary to be kept in the system after be ijoined with S. Furthermore, it is possible that another relation named R exists in the current database. The existing R should not be overwritten by the R shipped from "mimi". So when R is shipped to the local site, the run time system assign it a system generated temporary name.

Third, since the domain information and relational information are shipped accompanying with the user defined data, the run time system is able to check the agreement of the shipped domains with the existing domains having the same names. If they are in conflict with respect to each other, a system error is displayed. On the other hand, if there are not the same name domains in the receiver, the system will create new domains accordingly.

The protocol for shipping flat relations is shown in figure 4.8:

```
Rel
<rel.name> "|" <rel.rvc> "|" <rel.numtuples> "|" <rel.numattrs> "|" <rel.numsortattrs>
<rel.domains[0].name> "|" <rel.domains[0].type>
......
<rel.domains[i].name> "|" <rel.domains[i].type>      (i: rel.numattrs-1)
<line 1.item 1> "□" <line 1.item 2> "□" ..."□"<line 1.item k>      (k: rel.numattrs)
......
<line j.item 1> "□" <line j.item 2> "□" ..."□"<line j.item k>      (j: rel.numtuples)
END

Note: "□" is a delimiter "\F".
        rel.rvc is the type of a relation. "r" for relation, "v" for view and "c" for computation
```

Figure 4.8: Protocol for shipping flat relation

Figure 4.9 shows an example of the data stream shipped across the network.

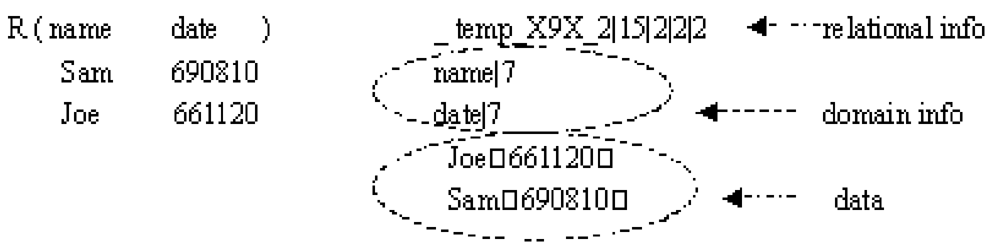


Figure 4.9: Example for shipping flat relation

Shipping Nested Relation

To begin with, we will explore a three level nested relation to understand how the data is organized and stored in JRelix for nested relations. Figure 4.10 is an example of nested relation.

Figure 4.11 shows the files about "faculty" on the disk:

```
faculty
  ( dept      profs                                      )
          ( name   office    course                    )
                           (  code    title            )
   CS      Merrett  304      612     database system
                             617     information system
           Newborn  306      767     E-commerce
                             431     Algorithms
   EE      Pat      412      530     Control System
                             538     Robot
```

Figure 4.10: Nested relation: faculty

Nested relations are relations whose attribute values may themselves be relations. The essence of nested relation is to subsume the relational algebra into domain algebra. To ship nested relations, we need to modify the algorithm. As discussed above, the algorithm for shipping flat relation is as follows:

1) ship relational information

2) ship domain information

3) ship data file

For nested relations, since relations are incorporated into domains, the second step should be revised to allow relations to be encapsulated in domains. Furthermore, the process is recursive to allow deep nesting. One more significant point is about surrogates. As shown in figure 4.11, surrogates are used for nested relations. These surrogates are system generated sequential number.

When the surrogates of a nested relation are shipped from the resource to the target, they may conflict with some of the existing surrogates at the target site. So the receiver will replace the original surrogates with the new surrogates generated by himself . Figure 4.12 shows the pseudo code for shipping flat or nested relation.

Figure 4.13 shows data stream transferred across network and Figure 4.14 shows files created at the receiver site for the assignment: `faculty <- pubA2/faculty`;

Faculty

dept	profs
CS	1
EE	4

.profs

.id	name	office	course
1	Merrett	304	2
1	Newborn	306	3
4	Pat	412	5

.course

.id	code	title
2	612	database system
2	617	information system
3	431	Algorithms
3	767	E-commerce
5	530	Control system
5	531	Robot

.rel

faculty□2□2□15□2
profs□3□4□15□0□
.course□6□3□15□0
...

.dom

code□7□1□0□
course□12□1□0
office□7□1□0□
dept□7□1□0□
profs□12□1□0□
title□7□1□0□
name□7□1□0□
...

.rd

faculty□dept□0□
faculty□profs□1□
.profs□.id□0□
.profs□name□1□
.profs□office□2□
.profs□course□3□
.course□.id□0□
.course□code□1□
.course□title□2□
...

Figure 4.11: Disk files and metadata about nested relation “Faculty”

```
SendRel (Rel) {
    // ship relational information
       send (<rel.name> | <rel.rvc> | <rel.numtuples> | <rel.numattrs>
           | <rel.numsortattrs>);
    // rel.rvc is the type of a relation. ``r'' for relation,
       ``v'' for view and ``c'' for computation

    // ship domain information
       for each domain[i] in the being shipped Rel do
       {
        if domain[i].type is a nested relation
           SendRel ( .rel.domains[i].name )       // recursive call
        else
           send (rel.domains[i].name | rel.domains[i].type )
       }

    // ship data file
      for each line in the data file do
      {
          read one line,
          send the line;
      }
}
```

Figure 4.12: Pseudo code for shipping flat or nested relation

4.3 Implementation details for distinct distributed data processing

4.3.1 Remote Assignment

The assignment operation assigns a "relation value" to a relation. There are two types of assignments in JRelix, normal assignment and incremental assignment. The former creates a new instance of the relation, while the latter adds the tuple data of the source relation to the assigned relation. In a distributed assignment, the source and/or the assigned relation might be remote relation . The source relation is not necessary an existing relation, in stead, it could be a temporary relation resulting from a relational expression. Any relations of the expression are allowed to be remote ones.

If the URL-based name structure, i.e. "aldatpheader" plus "element", only appears

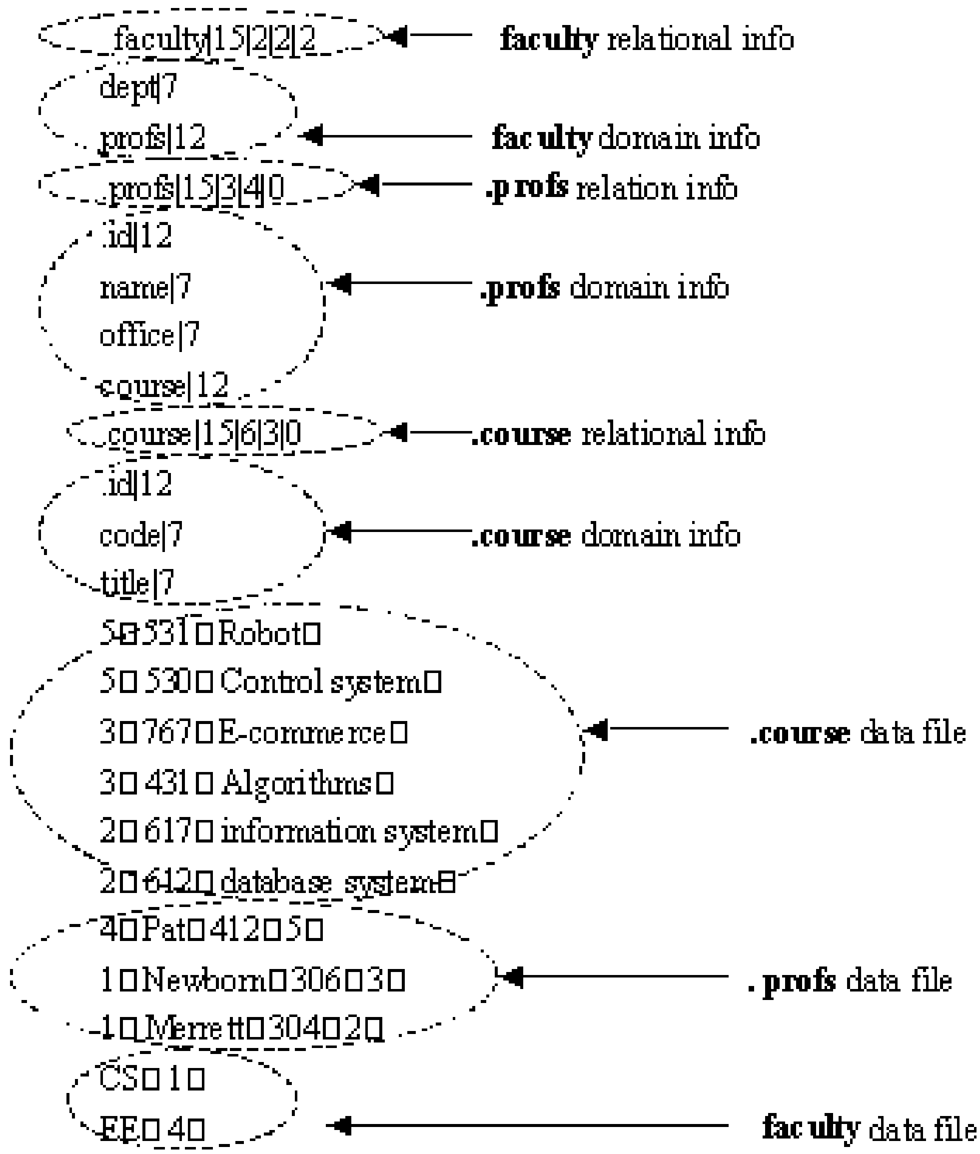


Figure 4.13: Data stream shipped for nested relation

Faculty

dept	profs
CS	14
EE	13

.profs

.id	name	office	course
13	Pat	412	10
14	Merrett	304	12
14	Newborn	306	11

.course

.id	code	title
10	530	Control system
10	531	Robot
11	431	Algorithms
11	767	E-commerce
12	612	database system
12	617	information system

Figure 4.14: Data files for relation "faculty" at the receiver site

at the right hand of the assignment, we call the assignment "right-hand assignment", else we call it "left-hand assignment". Table 4.5 gives examples of both right-hand and left-hand assignment.

Right-hand assignment:	A <- aldatp://mimi/~tim/pubA/S; A <- [name] where date = "661120" in aldatp://mimi/~tim/pubA/S; A <- aldatp://mimi/(R ijoin S); A <- T ujoin aldatp://mimi/([name] in (R ijoin S)); A <+ R ijoin aldatp://roo/pub/S;
Left-hand assignment:	aldatp://roo/A <- IntPerChg ijoin IntPer; aldatp://roo/A <- R ijoin aldatp://mimi/~tim/([name] in R ijoin S); aldatp://roo/A <+ aldatp://mimi/(R ijoin S);
Remote assignment statement:	aldatp://roo/pubA/{ A<- R ijoin S };

Table 4.5: Examples of remote assignment

For the right-hand assignment, whenever the interpreter encounters a node with non-null URL, it calls aldatpClient to throw the sub tree to the target site. The sub parser tree is either a single node corresponding to a remote relation or an expression tree. What ever it is, a response relation will be shipped back if no failures occur.

Having received the response relation, the interpreter continues the unfinished work. Finally, the source "relation value" is evaluated, and then is assigned to the local target relation. Since shipping parser tree and shipping relation are already introduced in Section 4.2, nothing is special for right-hand assignment.

For the left-hand assignment, the assigned relation is a remote one. If we allow the interpreter to treat the left-hand node the same as it treat the right ones, then the interpreter will attempt to fetch a remote relation to the local site and over write it instead of assigning source relation value to the remote target relation by mistake. So our solution is as follows:

First, because the parser can tell a left-hand assignment, a special operation type is used to distinguish left-hand assignment.

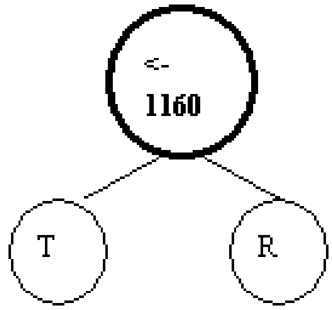
It can be seen from the Figure 4.15, the normal node type for assignment is 160. On the other hand, the node type for left-hand assignment is 1160. Thus, the interpreter can distinguish the left-hand assignment quite easily.

Second, after the interpreter telling the left-hand assignment from other assignments, it will evaluate the right hand expression as usual and get a temporary relation that is the relation value of the right-hand expression. Next, the interpreter constructs a remote assignment statement assigning the temporary relation to the target relation.

Example

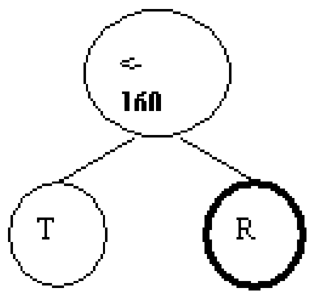
```
Suppose we execute the following statement at JRelix running at
‘‘roo:~zwang26/public_aldatp/pubA’’
     aldatp://roo/A <- R ijoin aldatp://mimi/~tim/( [name] in R ijoin S);
Step 1:  Distinguish left-hand assignment.
Step 2:  The interpreter evaluates the following expression,
           R ijoin aldatp://mimi/~tim/( [name] in R ijoin S)
         The result relation value is stored in a temporary relation, e.g. _temp_X9X_5
Step 3:  The interpreter builds a remote statement as following
           aldatp://roo/{ A <- aldatp://roo/~zwang26/pubA/_temp_X9X_5};
Step 4:  Interpret the statement created in step 3.
```

aldatp://localhost/T <- R;



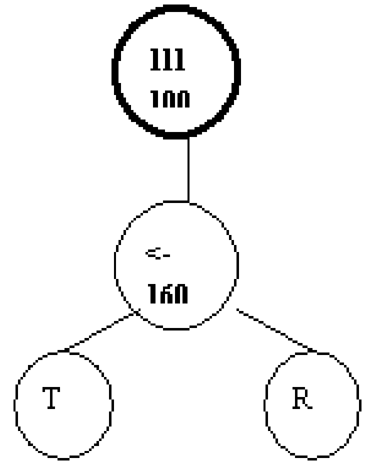
identifier	name	opcode	type	Num children	info	url
Assignment	null	161	**1160**	2	null	**aldatp://localhost/**
Identifier	T	230	230	0	null	**null**
Identifier	R	230	230	0	null	null

T <- aldatp://localhost/R;



identifier	name	opcode	type	Num children	info	url
Assignment	null	161	**160**	2	null	**null**
Identifier	T	230	230	0	null	null
Identifier	R	230	230	0	null	**aldatp://localhost/**

aldatp://localhost/{T <- R};



identifier	name	opcode	type	Num children	info	url
StatementBlock		111	100	1	null	**aldatp://localhost/**
Assignment	null	161	**160**	2	null	**null**
Identifier	T	230	230	0	null	null
Identifier	R	230	230	0	null	null

Figure 4.15: distinguish left-hand assignment

In summary, the rule to deal with left-hand assignment is evaluating relational expression locally and executing the assignment remotely.

4.3.2 Remote Update

Here are three cases of **update** statement.

Update R **add** S;

Update R **delete** S;

Update R **change** `<statements>` **using** S

S is a relation or any relational expression, and the using S clause (which is optional) in the change command uses the natural join of S with R to select the part of R that will be change. S may also be preceded by a join operator other than a natural join. The `<statements>` in this case are usually assignment statements changing values of attributes.

In all three cases, if R is a remote relation, the update statement is called left-hand update. If only S is remote relation, or if S is a relational expression and the expression consists of remote relations, then we call the update statement right-hand statement.

Right-hand update:	Update A add aldatp://mimi/~tim/pubA/S; Update A delete (T ujoin aldatp://mimi/(R ijoin S)); Update A change name <- Tom using ([name] where date =“661120” in aldatp://mimi/ tim/pubA/S);
Left-hand update:	Update aldatp://roo/A add (IntPerChg ijoin IntPer); Update aldatp://roo/A delete (R ijoin aldatp://mimi/~tim/([name] in R ijoin S)); Update aldatp://roo/A change date <-“031007” using aldatp://mimi/(R ijoin S);
Remote update statement:	aldatp://roo/pubA/ update A add (R ijoin S) ;

Table 4.6: Examples of remote update

For the right-hand update, the interpreter evaluates S as usual. Whenever it encounters a remote node when evaluating the expression, the interpreter calls aldatpClient to throw the sub parser tree rooted at that node to the target site. After the remote server has interpreted the parser tree and sent back the response relation, the interpreter continues the unfinished work. Finally, a “relation value” is evaluated, and then the local interpret uses it to do update. The idea is the same as interpreting right-hand assignment.

For the left-hand update, a special operation type is used to distinguish left-hand update. Since if we allow the interpreter to treat the being updated relation the same as it treat the right hand S, then the interpreter will attempt to fetch a remote relation to the local site and update it locally by mistake. So our solution is as follows:

First, the parser assigns the update node a special type when it detects a left-hand update. The normal node type for update is 180. On the other hand, the node type for left-hand update is 1180. Thus, the interpreter can distinguish the left-hand update .

Second, after the interpreter telling the left-hand update from other updates, it will evaluate the right hand expression as usual and get a temporary relation that is the relation value of the right-hand expression. Next, the interpreter constructs a remote update statement updating the target relation using the temporary relation.

Example

```
Suppose we execute the following statement at JRelix running at
"roo:~zwang26/public_aldatp/pubA"
      Update aldatp://roo/A change name <- "Luc"
             using R ijoin aldatp://mimi/~tim/( [name] in R ijoin S);
Step 1:  Distinguish left-hand update.
Step 2:  The interpreter evaluates the following expression,
           R ijoin aldatp://mimi/~tim/( [name] in R ijoin S)
         The result relation value is stored in a temporary relation, e.g. _temp_X9X_5
Step 3:  The interpreter builds a remote statement as following
           aldatp://roo/{ update A change name <- "Luc"
             using aldatp://roo/~zwang26/pubA/_temp_X9X_5 };
Step 4:  Interpret the statement created in step 3.
```

In summary, the rule to deal with left-hand update is evaluating relational expression locally and executing the update remotely.

4.3.3 Remote View

As introduced in Section 2.5, view does not hold data upon declaration and initialization. It is usually regarded as a functional definition. In JRelix notation, **is** replaces the assignment arrows, `<-` and `<+`. Thus,

V is R ; (or V is `< relational expression >`)

just defines V to be synonymous with R, and no evaluation is performed until a subsequent assignment, or other operation such as print forces it. Tuple data are generated on the fly.

Like assignment, if V is a remote one, the statement is called left-hand view. Else if "aldatpheader" only occurs at right-hand R or the relational expression, the statement is called right-hand view.

Right-hand view:	V is add aldatp://mimi/ tim/pubA/S;
Left-hand view:	aldatp://roo/V is IntPerChg ijoin IntPer; aldatp://roo/V is R ijoin aldatp://mimi/S;
Remote view declaration statement:	aldatp://roo/pubA/ V is R ;

Table 4.7: Examples of remote view

Right-hand view is similar to right-hand assignment except the evaluation of right side relational expression is deferred. Once the evaluation is invoked, the methods for evaluation are the same.

For the left-hand view, we use the similar idea for left-hand assignment and left-hand update, which is changing left-hand view to a remote statement of view declaration.

First, the parser assigns a special type when it detects a left-hand view. The normal type for view is 140. On the other hand, the type for left-hand view is 1140. Thus, the interpreter can distinguish the left-hand view quite easy.

After the interpreter telling the left-hand update from other updates, the interpreter constructs a remote statement of view declaration in remote site's point of view.

Example

```
Suppose we execute the following statement at JRelix running at
"roo:~zwang26/public_aldatp/pubA"
     aldatp://roo/V is R ijoin pubA1/R ijoin aldatp://mimi/~tim/S;
Step 1:  Distinguish left-hand view.
Step 2:  The interpreter builds a remote statement as following
          aldatp://roo/{ V is  aldatp://roo/~zwang26/pubA/R  ijoin
          aldatp://roo/~zwang26/pubA/pubA1/R  ijoin
          aldatp://mimi/~tim/S };
Step 3:  Interpret the statement created in step 2.
```

In summary, the rule to deal with left-hand view is changing left-hand view declaration to a remote statement of view declaration. All the right-hand stuffs are rewritten according to the new site's point of view.

4.3.4 Remote Computation

A computation can be thought of as a compressed relation, in which the relationship is given not explicitly by data but implicitly by code.

Take the computation IntPerChg introduced in Section 3.6 as example. Because a computation must be thought of as a relation, when the interpreter interpret the following statement, it will ask aldatpClient to ship the IntPerChg as if it is a relation.

```
Intint <- [p] where I=0.12 & i=0.01 in aldatp://localhost/IntPerChg;
```

In section 4.2.3, shipping data is introduced, but only regular flat and nested relation are involved. As we know now, a computation is a special kind of relation, thus shipping data for computation is actually shipping code. The algorithm shown in Figure 4.12 is modified to Figure 4.16

```
SendRel (Rel) {
    // ship relational information
       send (<rel.name> ‘‘|’’ <rel.rvc> ‘‘|’’ <rel.numtuples> ‘‘|’’
             <rel.numattrs> ‘‘|’’ <rel.numsortattrs> );

   // ship domain information
      for each domain[i] in the being shipped Rel do
      {
       if domain[i].type is a nested relation
          SendRel ( .rel.domains[i].name )           // recursive call
       else
          send (<rel.domains[i].name> ‘‘|’’ <rel.domains[i].type> )
      }

   // ship data file
      if ( Rel.type is relation ) {
         for each line in the data file do
         {
          read and send the line,
         }
      }
      else if ( Rel.type is computation ) {
         dump the syntax tree of the computation code into a flat text format;
         send the dumped tree;
      }
}
```

Figure 4.16: Pseudo code for shipping relation and computation

Figure 4.17 presents the data stream shipped across the network for computation

IntPerChg.

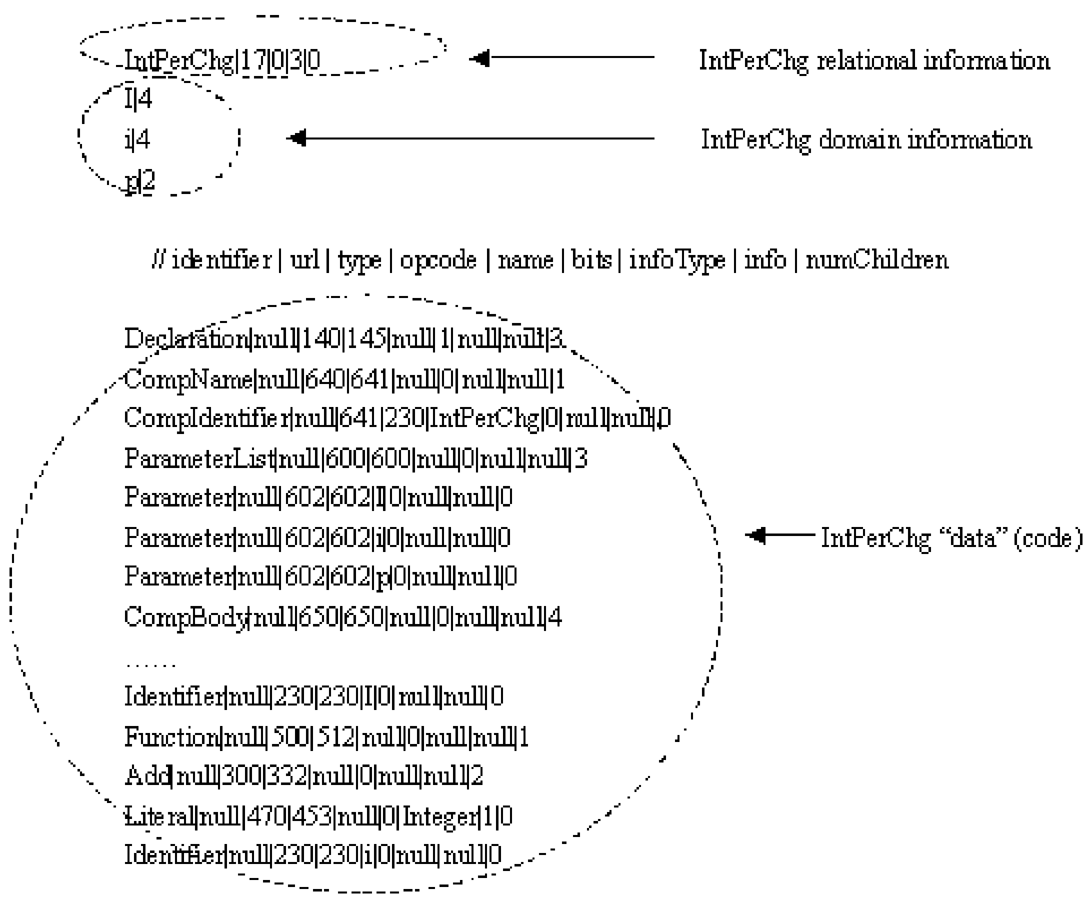


Figure 4.17: Data stream shipped for computation IntPerChg

The caller aldatpClient receives the data stream, creates a temporary relation or a temporary computation according to the relation type, which then is used by the interpreter or the computation processor.

4.3.5 Remote Computation Call

Computation calls are top-level computations. Basically they are similar to the procedure calls in some programming language. They could take relations as parameters, and output relations as the result of computation. These in or out parameter relations could be remote or local ones. The computation call itself could also be called locally or remotely.

Right-hand computation call:	Compcall (in aldatp://mimi/R, out R);
Left-hand computation call:	Compcall (in R, out aldatp://mimi/R);
Invoke computation call remotely:	aldatp://roo/pubA/ compcall (in R, out S) ; aldatp://roo/pubA{ compcall (in R, out aldatp://mimi/R) };
Ship computation code to local site, invoke it locally	aldatp://roo/pubA/compcall (in R, out S) ; aldatp://roo/pubA/compcall (in aldatp://mimi/R, out S) ;

Table 4.8: Examples of distributed computation call

In table 4.8, for both the first and the second case, the computation code is at local site and the computation is executed locally. For the third case, the computation is executed remotely. Remote statement execution will be introduced in section 4.3.6. For the fourth one, the remote code is shipped to the local site and invoked locally. Top-level computation code shipping is the same as normal computation code shipping, which is introduced in section 4.3.4

For each remote "in" parameter relation, the computation processor calls the aldatpClient to get the relation from remote server. The aldatpClient throws the one node parser tree to the target server, and a response relation will be shipped back if no failures happen.

For each remote "out" parameter relation, a temporary relation is generated first to hold the data. Then a remote assignment statement is constructed to assign the temporary relation to the target output relation.

Figure 4.18 shows the pseudo code for computation call.

4.3.6 Remote Statement Block and Command

The root node of the entire parser tree has a non-null URL, so the interpreter calls the aldatpClient to throw the whole tree to a remote target server. The server receives and interprets the parser tree, throws some part sub-trees further to other servers if necessary, and at last sends a response code back to the caller. This has already been introduced in section 4.2, so we are not going to elaborate any more.

```
Computation.applyInOut ()       // Apply this computation using the in/out syntax.
{
// bring in the input parameters
If the input parameter is a remote relation,
   call adltpClient to get the input relation;
else
   bring it in from the local calling environment ;

Execute the computation block ;

// output
If the output parameter is local relation
   Copy out the output variables to the calling environment ;

If the output parameter is remote relation {
   Copy out the output variables to the calling environment  with
       a system generated temporary name;
   Create and execute a remote statement to assign the temporary
       relation to the target output relation;
}
}
```

Figure 4.18: pseudo code for computation call

4.3.7 Left-hand Operations for Stand Alone JRelix

We have introduced four kinds of left-hand operations: left-hand assignment, left-hand update, left-hand view and left-hand computation call. The basic idea to deal with the left-hand operations is translating left-hand statement into remote statement.

Example.

```
For the following statement at JRelix running at
"roo:~zwang26/public_aldatp/pubA"
    aldatp://roo/A <- R ijoin aldatp://mimi/~tim/( [name] in R ijoin S);
```

```
Step 1 :  R ijoin aldatp://mimi/~tim/( [name] in R ijoin S) is evaluated and
          stored in a temporary relation, e.g. _temp_X9X_5
Step 2:  build and execute the following remote statement
         aldatp://roo/{ A <- aldatp://roo/~zwang26/pubA/_temp_X9X_5};
```

If the JRelix running at "roo:~zwang26/public_aldatp/pubA" is a stand alone JRelix, i.e. it is not a aldatp server. It is obvious that the statement created in step 2 does not work. So we need to change the solution a little bit for stand-alone JRelix as follows.

```
Step 1' : R ijoin aldatp://mimi/~tim/( [name] in R ijoin S) is evaluated and
           stored in a temporary relation, e.g. _temp_X9X_5
Step 2' : Ask the JRelix server running at "roo: ~/public_aldatp" to give a
           system generated temporary name. E.g. _temp_X9X_15
Step 3' : Ship the relation _temp_X9X_5 from "roo:~zwang26/public_aldatp/pubA" to
          "roo:~/public_aldatp".
          The shipped relation has a new name _temp_X9X_15 at "roo:~/public_aldatp".
Step 4':  Build and execute the following remote statement
          aldatp://roo/{ A <- _temp_X9X_15};
```

The same idea is used in left-hand update, left-hand computation call for stand-alone JRelix. The exception is left-hand view, since view defers the evaluation until later. We can't send the result of the evaluation to the opposite side upon view declaration. As a result, left-hand view is not allowed for stand-alone JRelix.

4.4 System Administration

4.4.1 Start Options

The start options of JRelix multidatabase system have already been illustrated in Section 3.10. Here, we summarizes them as follows:

```
1) Root level server
Start ``java aldatpTLd &'' at a ``public_aldatp'' directory
```

```
or
Start ``java JRelix'' at a ``public_aldatp'' directory
2) Lower level JRelix server
Start ``java JRelix'' at any sub directory under ``public_aldatp''
3) Protected JRelix server
     Start ``java JRelix'' at a directory outside ``public_aldatp''
4) Stand alone JRelix
     Start ``java JRelix -SA'' at any directory
```

Start aldatpTLd

On each computer, the root level server manages and coordinates all the JRelix servers on that machine to carry out the distributed tasks. If the root server is not started on one machine, all the JRelix running on that machine are **stand alone** JRelix systems, i.e. they are not servers. The root server is responsible for listening to the "well-known" port number and responding to the requests from JRelix systems on the local machine or any other computers linked by the Internet. Meanwhile, it maintains a system dictionary about the information on port numbers and the current lower level servers.

The aldatpTLd doesn't launch a JRelix session at all. No user data and system tables are loaded at this time. The parser, the interpreter, and the execution engine are not invoked for the moment. Figure 4.19 shows the flow chart.

Start JRelix Instance

Four distinct start options are obtained from the same java program: JRelix.java. Figure 4.20 shows the control flow of selecting an option.

Among the four options, the **stand alone** option is the simplest one. It starts a session with front-end interface, by which end-user can enter JRelix commands. Meanwhile, the main loop of the interpreter is invoked. In the loop, the interpreter repeatedly calls the parser to translate user input command into syntax tree, receives syntax tree passed from the parser, traverses the syntax tree and decomposes it into

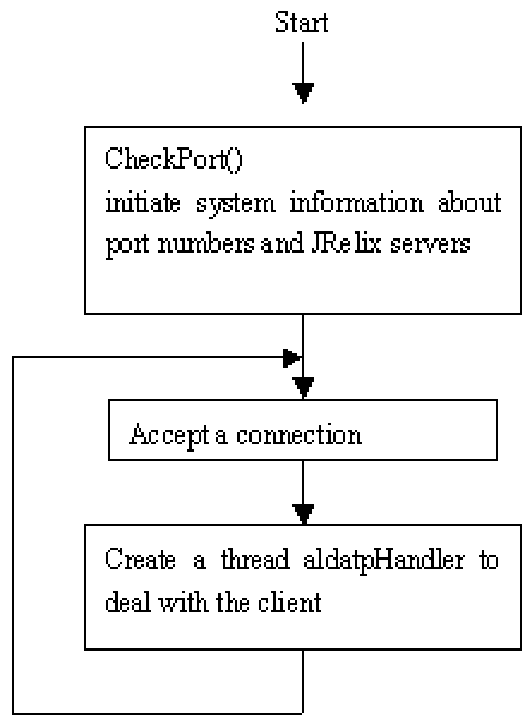


Figure 4.19: The main loop of aldatpTLd

a set of method calls executed by the execution engine.

Lower level server and **protected server** are basically the same. They not only start a front-end interface session to process end-user input commands, but also start a server. The server repeatedly listens to a specific port, accepts connections and creates multiple threads to deal with the requests from clients. These requests are passed from other JRelix systems. A typical request is to interpret a syntax tree and return a result as shown in the previous sections in this chapter. There are other kinds of requests to a JRelix lower level server or protected server, which will be discussed in the latter sections.

As mentioned in section 3.10, at one moment only one root server can be launched at one machine. The root server is either an aldatpTLd or a JRelix running at a "public_aldatp" directory. The **JRelix root server** (not the aldatpTLd) plays two roles. It has all the functions of a lower level server, as well as functions of a root server, aldatpTLd. It starts a front-end interface session to process end-user input commands. It also starts a special server. This server not only manages and coordinates other JRelix servers on that machine to carry out the distributed tasks, but also deals with data processing requests such as interpreting a syntax tree asked from other JRelix.

4.4.2 Manage Port Numbers

Apart from the “well-known” port number consumed by the root server, each lower level server also consumes one port. However, it is unwise to make all port numbers for each member in the multiple system to be public “well-known”. In our implementation, a global master manages these ports and these ports are totally transparent to end-users.

The port numbers reserved for lower level servers are written into a disk file “.ports”. This file is created manually by a system administrator. Each line in the “.ports” is a port number”, and this file must be already existent at the same directory at which the root server will be started.

Figure 4.19 gives a brief process flow of aldatpTLd. The beginning step is initiating system administration information.

When the root server is being started, all the port numbers in the “.ports” are loaded into RAM. Although these ports are reserved for aldatp lower level servers usage and ideally all of them are available when being allocated, we still check the availability of each port in case some of them have already been used by other applications. Those still available ports are pushed in to a stack.

This check brings us another benefit. Suppose the root server shuts down (e.g., the root JRelix is exited or the aldatpTLd is killed by user) while many lower level servers are still running, each consuming a port number. Under this circumstance, the root server can be started without restarting all the running lower level servers. Because if the root server detects a port is being used, it will assume the port is being used by a lower level server and then ask the opposite side for its identifier, by

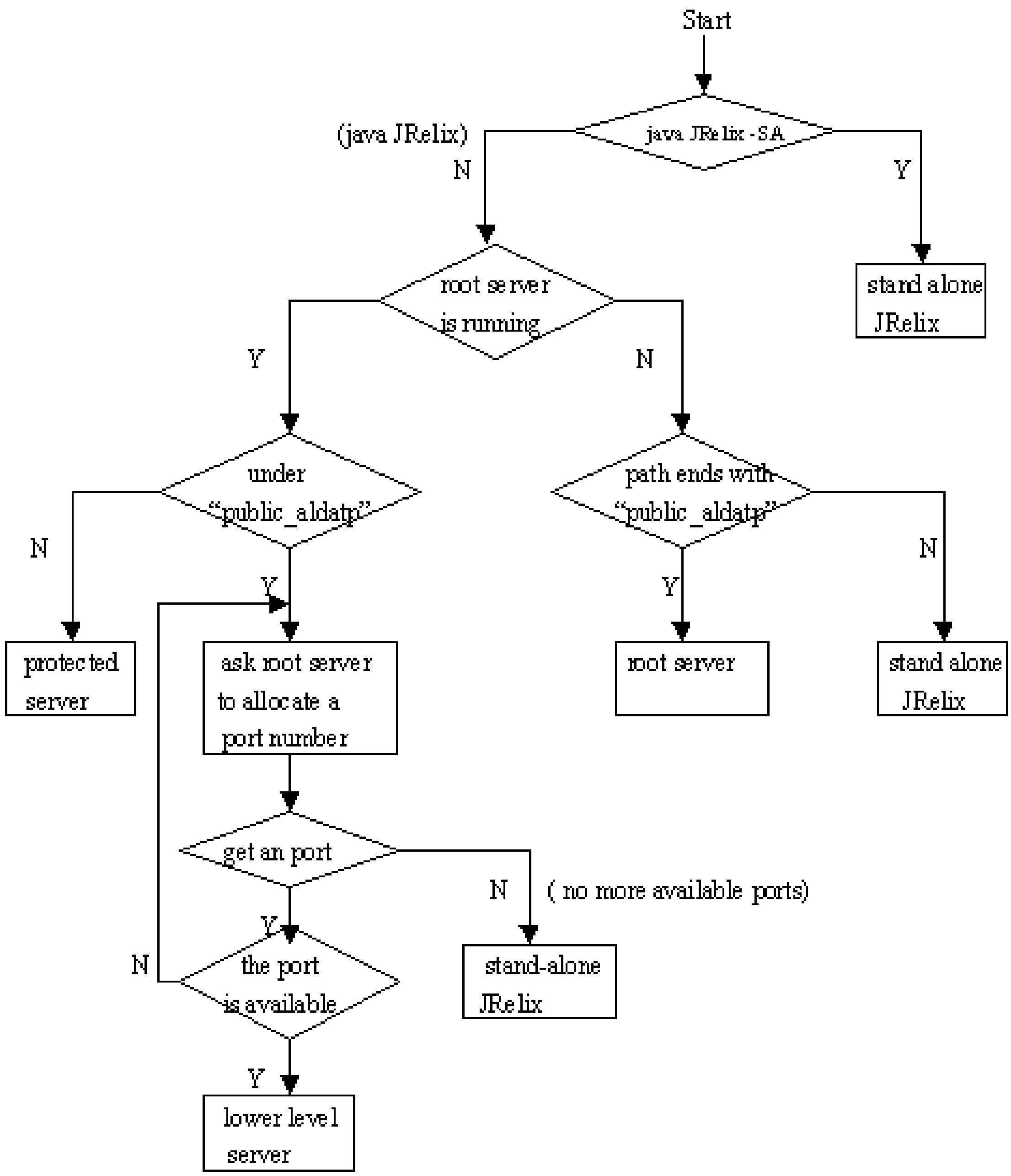


Figure 4.20: JRelix start options

sending the message: “checkPort”.

If the port is really occupied by a lower level server, the lower level server understands the protocol, so it returns its identifier information according to a pre-defined protocol:

```
ALDATP_PATH
<path>
```

In this case, the root server writes the identifier and corresponding port number into a hash table. If no expected result is received, the root server will regard this port as being used by other applications.

In short, the root server maintains a stack to store available ports and a hash table to keep the information about current running lower level servers and protected servers.

So far, the root server is running. Whenever it accepts a connection, it creates a thread to deal with it. The requests processed by root server are summarized in Figure 4.21. Detail descriptions are given following the table.

Note: Figure 4.21 gives all the possible requests to the root server. Figure 4.24 summaries all the possible requests to a normal lower level server in section 4.4.3.

1. allocatePort

When a lower-level server or protected server is being lunched, it asks the root server to allocate an available port. Figure 4.22 shows the flow chart for “allocatPort”.

If no ports are available, the “chekPort” is invoked. If some ports that were used by other applications while the root server was starting are released at this moment, then these ports are pushed into the satck ”availablePort” so that they can be allocated. Moreover, those new ports added manually to the “.ports” file after the root server’s starting can also be pushed into the stack “availablePort” by the ”checkPort”.

Since the current JRelix is RAM-based and has no concurrency control mechanism, it is not allowed to start more than one JRelix server concurrently at the same directory. The hash table “ports” maintained by root server is helpful to prevent users from starting duplicate servers.

Request	Caller	Description	Response
allocatePort <user name> <user home directory> <path>	JRelix or JRelixBack (see § 4.4.3)	Ask root server to allocate an available port to start a lower level or protected server	OK <port> *or* NO available port *or* JRelix is running on this directory
confirmPort <path> <port>	JRelix or JRelixBack	Confirm the port is OK	Root server register it into the hash table “ports”
quit <path>	Interpreter or aldatpHandler	The lower level or protected server notifies the root server when it is going to quit.	Root server deletes the entry from hash table “ports”, adds the port to the stack “availablePort”
enquirePort <url>	aldatpClient	Ask the root server for the port number of the url . If the path corresponding to the url is found in the hash table	OK <port>
		If not found, try to start a background server at that path. If start successfully	start server <port>
		otherwise	server not available

Figure 4.21: Requests processed by root server

2. confirmPort

Initially, the “availablePort” stack stores the ports available at the moment the root server was being launched. It is not guaranteed that these ports are still available when they are being allocated to some lower level servers. If the port be allocated can be used to create a server, the lower level server will confirm this port number, thus the root server adds the lower level server’s path and the port number to the hash table “ports”. Otherwise, the caller will call the root server to allocate a port once more.

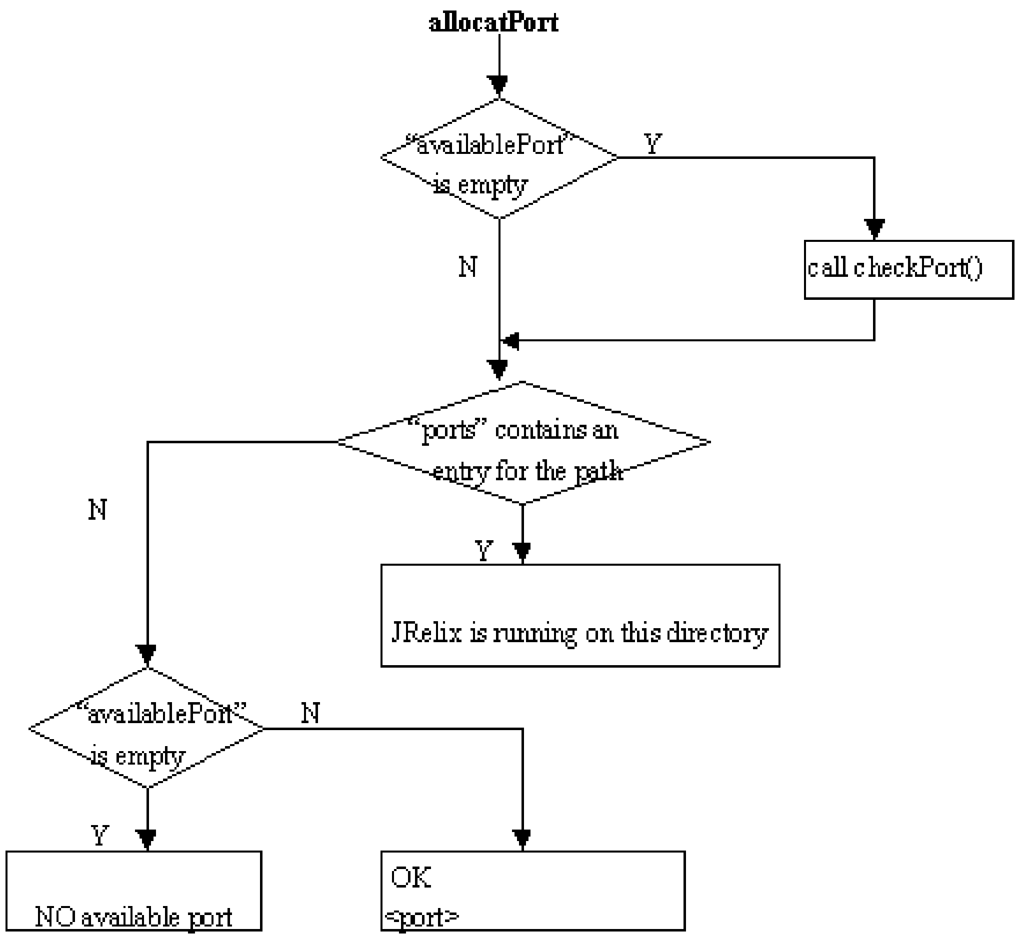


Figure 4.22: Flow chart for allocatePort

3. quit

The "quit" request is simple. The root server deletes the entry in the hash table "ports" and adds the free port to the stack "availablePort"

4. enquirePort

Figure 4.23 shows the flow chart for "enquirePort".

When an "aldatpClient" receives a request from the Interpreter, it extracts the URL from the root node of the sub-tree. From analyzing the URL, it gets the host name and path information about the target JRelix. Then it connects with the root

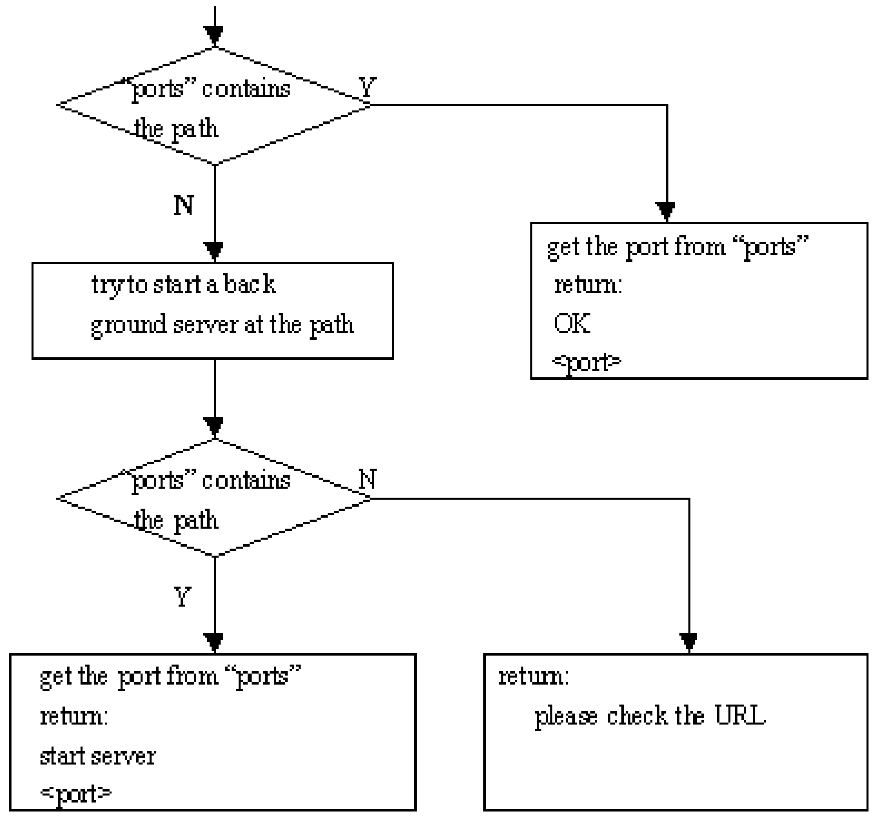


Figure 4.23: Flow chart for dealing enquirePort

server on that host, enquires the port number for the JRelix server running at the target path. If there is no entry for the path in root server's system table, the root server will try to start a background server and return the port of the background server to the caller. If the path doesn't exist, the system will warn the user to check the URL.

4.4.3 Background Server

As shown in Figure 4.23, background servers are automatically started by the root server if necessary. This greatly decreases the human intervene and increase the autonomous capability. Unlike normal lower level server or protected server, a background server does not start a front-end interface session to process end-user input commands, but only start a background daemon. The daemon repeatedly listens to a specific port, accepts connections and creates multiple threads to deal with the requests from clients, e.g. to interpret a syntax tree and return a result.

For JRelix with an end-user interface, the user will finally terminate the session by typing “quit;” at a JRelix command line. However, the JRelix background server is started automatically by the root server, therefore it is totally transparent to end-users. For example, a user named Joe starts a JRelix session and inputs the following statement:

```
aldatp://roo/~tim/{ A <- R ijoin S};
```

Joe does not care whether or not a JRelix server has already been launched at “roo:~tim/public_aldatp”. If a server is running there, no matter it is a background or front-end one, Joe uses it for free. If the JRelix started by Joe ask root server for the port number of JRelix running at “roo:~tim/public_aldatp”, the response code from the root server is

```
 OK
<port>
```

In contrast, if no server is running there, then the root server will start a background server at “roo:~tim/public_aldatp” for Joe, but Joe is not aware of it. The response code from the root server to the JRelix started by Joe is

```
start server
<port>
```

So the JRelix started by Joe knows the background server running at “roo:~tim/public_aldatp” is initially started for him. Although the background server can be invoked by anyone else afterwards, the JRelix started by Joe has the responsibility to stop that background server when Joe quits his session. Otherwise the background server will potentially never be stopped unless the system crashes or the machine is shut down.

Each JRelix system maintains a system table that keeps records of other JRelix background servers initially started for it. When it is stopping, it notifies every background server in the table to terminate.

When a background server receives a "quitServer" request, the server stops creating new threads for new connections. After all the active threads finishing the works at hand, the background server stops eventually.

Up till now, all the possible requests to the normal lower level server are introduced. Here we summarize them in figure 4.24

4.4.4 Security Issues

Protected Server

The JRelix systems running outside "public_aldatp" are private and protected. In general, only the owners can operate these systems at local sites. One exception is "left-hand view". For example, the JRelix running at "roo:~zwang26/priv" declares the following view,

```
Aldatp://roo/~tim/V is R ijoin S;
```

When this view is invoked, the JRelix running on "roo:~tim/public_aldatp" would access data from "roo:~zwang26/priv" to actualize the view. So the "roo:~zwang26/priv" must posses the sever ability to response this data access request.

As illustrated in section 4.2.1.1, the URL-based name structure is as follows:

```
‘‘aldatp://’’  ((<LETTER>)+ (<DOT>)*)+ <SLASH>
               ( <TILED> <IDENT> <SLASH>)* (<IDENT> <SLASH>)*
```

header + host name + alias of user home directory (optional) + rear part of the target path immediately after "public_aldatp"(optional)

When a URL is translated into a path, the "public_aldatp" is added into the result, e.g.

Request	Caller	Description	Response
checkPort	aldatpTLd	Response root server its path	ALDATP_PATH <path>
ParserTree <dumped node 1> …… <dumped node n> END	aldatpClient	AldatpServer rebuild the tree and call local interpreter to interpret it. Return the response code. If the tree is an expression, send back the result relation	<response code> Rel …… / see Figure 4.9 END
quitServer	Interpreter	The lower level server or protected server notifies the JRelix background servers, initially started for him, to exit.	
getRelName	aldatpClient	The stand alone JRelix ask a JRelix server for a temporary relation name before it send a relation to the server.	<name>
PutRel …… // see Figure 4.9 END	aldatpClient	The stand alone JRelix send a relation to a JRelix server	Jrelix server build the relation

Figure 4.24: Requests processed by lower level or protected server

```
aldatp://roo/~tim/R : relation R of the JRelix running at ~tim/public_aldatp on ``roo''.
aldatp://roo/~tim/pubA/R : relation R of the JRelix running at ~tim/public_aldatp/pubA
                           on ``roo''.
```

So the URL-based name structure can only refer to the elements of the databases running under "public_aldatp". Suppose we start a JRelix server at a directory outside "public_aldatp", this server will never be pointed to by our URL-based name structure. So we almost get the "protected" characteristic for free.

To make the left-hand view work, we use a special URL-based name structure. Instead of using "aldatp://", we use "aldatpProtect://" as the header.

Example:

```
 Suppose we execute the following declaration at ``roo:~zwang26/priv''
   aldatp://roo/~tim/V is R;
Step 1:  Detect the statement is a left-hand view and the JRelix itself is a protected
         JRelix
Step 2:  The interpreter builds a remote statement as following
         aldatp://roo/~tim/{ V is  aldatpProtect://roo/~zwang26/priv/R };
Step 3:  Interpret the statement created in step 2.
```

When this view is actualized, the interpreter encounters a node with non-null URL. It calls aldatpClient to process the sub tree rooted at this node. The aldatpClient finds the URL header is "aldatpProtect", as a result, it translates the URL into a target path without adding "public_aldatp". In this e xample the aldatpClient extracts host name "roo" and target path "~zwang26/priv" from the URL. Then it connects with the root server running on "roo" and gets the port number for JRelix running at "~zwang26/priv". Finally, the aldatpClient communicates with the JRelix running on "roo:~zwang26/priv" directly.

"aldatpProtect" is designed for interior URL-based name structures. It is totally transparent to end-users. It is an illegal URL header for the parser. This guarantees the protected server is isolated from malicious intruders and curious users.

For the left-hand assignment and left-hand update, we first evaluate the right-hand expression and store it in a temporary relation. Then we convert the left-hand operation into a remote operation. The temporary relation will be read while the remote assignment or update is being interpreted. To make the temporary relation readable from remote sites, we use "aldatpProtect" as the aldatp header. The same idea can be used for left-hand computation top-level call.

Example

```
Suppose we execute the following statement at JRelix running at ``roo:~zwang26/priv''
aldatp://roo/A <- R ijoin aldatp://mimi/~tim/( [name] in R ijoin S);
Step 1:  Distinguish left-hand assignment and the JRelix itself is a protected JRelix
Step 2:  The interpreter evaluates the following expression,
           R ijoin aldatp://mimi/~tim/( [name] in R ijoin S)
         The result relation value is stored in a temporary relation, e.g. _temp_X9X_5
Step 3:  The interpreter builds a remote statement as following
           aldatp://roo/{ A <- aldatpProtect://roo/priv/_temp_X9X_5};
Step 4:  Interpret the statement created in step 3.
```

An alternative solution is using the way described in section 4.3.7. Instead of waiting the opposite side to ask for reading the temporary relation, it can send that temporary relation to the opposite side on its own initiative.

File Access Permission

In general, the JRelix under "public_aldatp" are publicly available, while the JRelix outside "public_aldatp" are private and protected. In this way, we control the access permission at a database or directory level. However, we are able to control the access permission more finely at a relation or file level.

The current JRelix system is RAM-based. The meta data (or system tables) are stored in permanent files on hard disk as well as memory in RAM. Once the interpreter starts, system tables are constructed in RAM from loading meta data files on disk. The system tables reside in RAM during the run-time till the JRelix session is terminated. Subsequently, they are written back to disk. However, the user-defined data are stored in permanent files on disk and read fully (assume they are small enough) into RAM only when referred. If the relation loaded into the RAM is going to be updated (including incremental assignment), the original data file on the disk is deleted. The JRelix run-time system generates new relational data in RAM. Finally, it recreates a data file with the same name as the original one on the disk.

In the lifetime of a user-defined relation, mostly it stays on the disk. We can change the mode of the data files manually by using Unix "chmod" command. Next time the data file is referred, the access permission to that relation can be different. In our system, we decide that only if the absolute mode of the relation data file permits read by others, can that relation be read by other JRelix systems; only if the absolute mode of the relation data file permits write by others, can that relation be written by other JRelix.

An aldatpHandler receiving a parser tree from a remote aldatpClient doesn't interpret the parser tree until it has checked the access permission. It traverses the tree, checks the mode of the data files for each relational identifier in the tree. If the relational identifier appears at the left side of a statement, the aldatpHandler checks its write permission for others. Otherwise, the aldatpHandler checks its read permission for others.

Example

Suppose we execute the following statement at "roo:~zwang26/public_aldatp/pubA"

```
R <- aldatp://roo/~tim/([name,date ] in (R ijoin S ijoin aldatp://mimi/~tim/T));
```

The aldatpHandler of JRelix running at "roo: ~tim/public_aldatp" receives the following parser tree.

It traverses the tree, and encounters five identifiers name, date, R, S and T. name and date are domains. T is a remote relation. Therefore R and S need to be checked. If both the disk files named R and S respectively permit read by others, the check procedure returns OK. Otherwise, it complains "permission denied to read XXX" .

Example Suppose we execute the following statement at "`roo:~zwang26/public_aldatp/pubA`"

```
Update aldatp://roo/~tim/R add ([name,date ] in (R ijoin S ijoin aldatp://mimi/~tim/T));
```

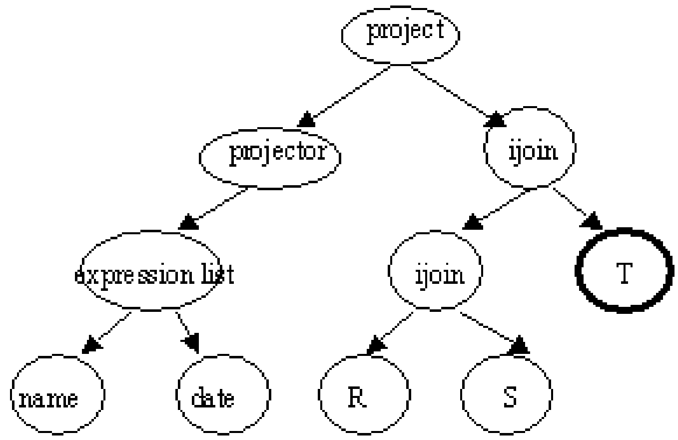


Figure 4.25: Syntax tree for expression: ([name,date] in (R ijoin S ijoin aldatp://mimi/ tim/T)

```
Step 1: The interpreter of JRelix running at "roo:~zwang26/public_aldatp/pubA"
        evaluates the expression: ([name,date ] in (R ijoin S ijoin aldatp://mimi/~tim/T)).
        Stores the result relational value into a temporary relation, e.g. _temp_X9X_5.
        Then builds and interprets the following remote statement
         aldatp://roo/~tim{ update R add aldatp://roo/~zwang26/pubA/_temp_X9X_5};
Step 2: The aldatpHandler of JRelix running at "roo: ~tim/public_aldatp"
         receives the parser tree shown in figure 4.25.
```

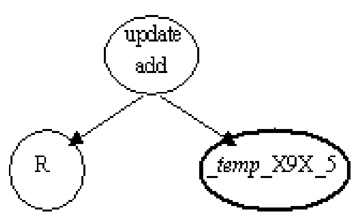


Figure 4.26: Syntax tree for : update R add aldatp://roo/z̃wang26/pubA/_temp_X9X_5

It traverses the tree, and encounters two identifiers R and `_temp_X9X_5`. `_temp_X9X_5` is a remote relation. Only R needs to be checked. R appears at the left side of the statement. The check procedure checks write permission for the file named R on the disk.

As mentioned before in this section, if the relation loaded into the RAM is going to be updated (including incremental assignment), the original data file on the disk is deleted. Afterwards, a data file with the same name as the original one is created on the disk. The potential problem is that the newly recreated file in Unix system has a default mode according to users' profile. As a result, the original mode is replaced with the default one, which is not desired. To avoid this situation, we keep record of the file mode before the file is deleted, and change the mode back to the original one when the file is recreated.

For left-hand assignment, if the assigned relation does not exist before, a new data file is created. Since this relation is created by remote JRelix system, we grant it a mode of permit both read and write to others. On the other hand, if the same name relation exists and it is forbidden to overwrite it by others, the check procedure returns "permission denied to write XXX". If the same name relation data files exists and it is allowed to overwrite it by others, a new relation data file is generated to replace it. The new relation data file has a mode permitting both the reading and writing to others.

Chapter 5

Applications with Aldatp

This chapter presents possible extensions with Aldatp in JRelix. In section 5.1, we introduce an application of event-based distributed system. In section 5.2, we suggest solutions to achieve location and fragmentation transparency.

5.1 Distributed event-based systems

Networking technologies and products now enable a high degree of connectivity across a large number of computers, applications, and users. In these environments, it is important to provide asynchronous communications for the class of distributed systems . This requirement has been filled by distributed publish-subscribe systems. In the RMI mode, a method in the remote interface of a particular object is invoked synchronously, i.e. the invoker waits for reply. In the event-based systems, notifications are sent asynchronously to multiple subscribers whenever a published event occurs at an object of interest.

Distributed event-based systems extend the local event model by allowing multiple objects at different locations to be notified of events taking place at an object. They use the publish-subscribe paradigm, in which an object that generates events publishes the type of events. Objects that want to receive notifications from an object that has published its events subscribe to the types of events that are of interest to them.

When a publisher experiences an event, subscribers that expressed an interest in that type of event will receive notifications.

Applications that communicate through a publish and subscribe paradigm require the sending applications (publishers) to publish messages without explicitly specifying recipients or having knowledge of intended recipients. Similarly, receiving applications (subscribers) must receive only those messages that the subscriber has registered an interest in.

As it will be illustrated, distributed event-based systems can be easily built with the Aldatp facility in JRelix. In this section, we explain how to build distributed event-based system in JRelix with a publish-subscribe example.

Create a bulletin board

To build a publish-subscribe system, firstly an empty global bulletin board is created at a global-known site. The bulletin board is a relation having two attributes.

```
>domain event string;
>domain subscribeBook string;
>relation Bulletin(event, subscribeBook);
```

An example of existing bulletin board with tuples is as follows:

```
Bulletin (  event                 subscribeBook   )
        post:add:conferences     aldatp://roo/conferSubscriber
        post:change:maps         aldatp://mimi/~tim/mapSubscriber
```

event is the name or identification of a published event type. Corresponding to each type of events, a subscribe book (a relation which holds user's information) exists at the site of the events taking place. The attribute *subscribeBook* gives names of these relations.

Suppose this bulletin relation is located at "roo:pubA" in the example.

Publish events

Suppose McGill University uses this system to publish its news to the interested public. News are stored in a relation named McGillnews, e.g.

```
McGillnews ( date         title      content ).
           2002-10-25   Top 100      ......
```

Whenever new news is added into, it will be sent to every one expressing an interest. Assume the McGill University news publish system runs at "`mimi:~mcgill/pubA`".

Step 1 create a subscribeBook

```
>domain subscriber string;
>relation newsSubscr(subscriber);
```

newsSubscr is a unary relation with only one attribute *subscriber*.

Step 2 create an event handler

```
>comp post:add:McGillnews ( ) is
{
 Loop <- newsSubscr;        // copy "newsSubscr" to "Loop",
 while [ ] in Loop          // while there is some tuples in "Loop"
 {
  S <- pick Loop;           // pick one tuple from "Loop" and assign it to "S"
  update Loop delete S      // delete the picked tuple from "Loop"
  eval S <+ New;            // "New" is the system generated relation in event handler,
                                it stores new added or changed tuples
                            // "eval" is an operator of relation metadata.
                               "eval S" is a relation whose name is the value of "S"
                               "S" is a singleton and a unary relation.
                               ("eval <Rel>" is not implemented yet.)
 }
};
```

Step 3 publish event

```
>relation eventInfo (event, subscribeBook) <-
    { ("post:add:McGillnews","aldatp://mimi/~mcgill/pubA/newsSubscr")};
>update aldatp://roo/pubA/bulletin add eventInfo    //add event into bulletin
```

Now, the bulletin looks like

```
Bulletin (  event                 subscribeBook   )
        post:add:conferences     aldatp://roo/conferSubscriber
        post:change:maps         aldatp://mimi/~tim/mapSubsrciber
        post:add:McGillnews      aldatp://mimi/~mcgill/pubA/newsSubscr
```

Subscribe

A user at Jrelix running on "roo: zwang26/pubA" checks the bulletin board

```
>pr aldatp:// roo/pubA/bulletin;
```

If she is interested in McGill news, then she subscribes to the event

```
>relation myInfo(subscriber) <- {("aldatp://roo/~zwang26/pubA/news")};
>update aldatp:// mimi/~mcgill/pubA/newsSubscr add myinfo;
                              // add into subscribe book
```

Now, the McGill news subscribe book looks like

```
newsSubscr(  subscriber                         )
             aldatp://roo/~zwang26/pubA/news
```

Events and Notifications

Whenever a new news is added into mimi :`~mcgill/pubA/McGillnews`, it is also automatically added into roo:`~zwang26/pubA/news`.

If the user "zwang26" is not interested in McGill news any more, she can delete her subscription

```
>update aldatp:// mimi/~mcgill/pubA/newsSubscr delete myinfo;
```

5.2 Seamless Distributed Database Systems

As we know, the true, generalized, distributed database systems are different from systems that provide remote data access. In a remote data access system, the user is able to operate on data at a remote site, or even on data at several remote sites simultaneously, but "the seams show"; the user is definitely aware, to a greater or lesser extent, that the data is remote, and has to behave accordingly. In a true distributed database system, by contrast, the seams are hidden. Full support for distributed database makes the distribution transparent to users. A single application should be able to operate transparently on data that is spread across a variety of different databases as if the data were all managed by a single DBMS running on a single machine.

Replication transparency and DBMS transparency are major research fields in Distributed Systems. Middleware systems try to overcome the heterogeneity faced when data is dispersed across different data sources. It would be nice if the DBMS instances at different sites could all support the same interface and participate in a distributed system. In other words, the ideal distributed system should provide DBMS independence.

DBMS transparency is out of the scope of this thesis. However, it is desirable to make our implementation to achieve some forms of transparency as we can.

- Fragmentation transparency

  Each fragment of a data item also has a global unique identifier. By maping the simple alias to complete names, physical fragmentat locations are hidden. If the query request is stated in terms of the unfragmented item name, the original data item needs to be reconstructed from its fragments.

  Distributed view management is provided in our implementation. Views can be defined using relations which are not local to the view definition site. Since views can simulate fragmentation, we can reconstruct the original relation from it fragments by using distributed view. For vertical fragmentation, a view can be definied at each site which take natural join of these fragements. Similarly, for horizontal fragmentation, a view can be definied at each site which take union of these fragements.

  Take the example illustrated in figure 1.1. Suppose an *employee* relation is horizontally fragmented into *Employee1* at “roo:˜tim/pubA and *Employee2* at "mimi:˜zwang26/pubB". *Employee* is the view taking unoin of *Employee1* and *Employee2*.

>Employee **is** aldatp://roo/˜tim/pubA/Employee1 **ujoin**

aldatp://mimi/˜zwang26/PubB/Employee2;

If the user's query is

```
>Ans <- [ id, name] where  salary=5000 in Employee;
```

Employee is reconstructed from fragments *Employee1* and *Employee2*.

- Location transparency

  In a URL-based name structure, each database element generated is prefixed with a site identifier. The site identifier is host name plus path. Since each site has a unique identifier, this approach ensures that no two sites generate the same name. No central control is required. However, this solution fails to achieve network transparency, since site identifiers are attached to names.

  To overcome this problem, we can create a set of alternative names or aliases for database elements. A user refers to an element by using the simple alias. This alias is then translated into the complete name with URL-structure.The alias and local relation names must be unique. With aliases, users will be unware of the physical location of data.

  A straightforward way to do so is using remote view. For instance, suppose R is the alias for a remote relation "roo: tim/pubA/R". By using the remote view,

```
>R is aldatp://roo/~tim/pubA/R;
```

  we can obtain the location transparency for "roo: tim/pubA/R".

  Or, the maping of alias to complete name is maintained in a *mapping table* (system meta data) at one site and then be sent to other sites. The system will look for the required identifier in the mapping table first. If it is not found, then the local relation table will be searched.

Chapter 6

Conclusions

This chapter begins with a summary of the work that has been accomplished. It concludes with suggestions for future enhancements.

6.1 Summary

We have built a URL-based name extension to a database programming language which gives it collaborative and distributed capability over the Internet.

Sharing resource is a main achievement of implementing Aldapt. In addition, some other basic objectives for distributing databases listed in Chapter 1 are accomplished in our implementation.

- Site autonomy is achieved. Each site is able both to control access from other sites to its own data and to manipulate its data without being conditioned by any other site. The system is able to grow incrementally and to operate continuously, with new sites joining to existing ones, without requiring existing sites to agree with joining sites on global data structures or definitions. There is not any reliance on a central site for central services .

- Performance transparency is accomplished. Commands or statements used to perform a task are independent of both the location of the data and the system on which an operation is carried out. The performance is independent of submission site

- Distributed view management is provided. Views can be defined using relations which are not local to the view definition site. Since views can simulate fragmentation, potentially both vertical and horizontal fragmentation are supported.

Moreover, the system obtains the following desirable features

- Remote procedure call or remote method invocation is achieved, as well as remote statements and commands execution are supported.
- Sites need not be geographically distant: different sites can be on the same computer. This is considered important not only for the development and testing of the database applications, but also for operational systems for security, accounting, or performance reasons.
- Security management is implemented. Data accessing permission can be controlled at both database (or directory) level and relation (or file) level. Database can be in either private or public mode. Changing mode is possible and easy to do. In fact, it is as easy as moving directory in an operating system. Public mode database can be launched in a "stand-alone" type or in a "publicly available" type.

Although it is desirable to give as many forms of transparency as possible, our implmentation does not aim to replace or compete with DBMS. Our implementation is not driven by all the transparency forms listed in Chapter 1.6. Basically speaking, we implement a mechanism for integreting distributed data processing and Internet capability into a databse programming language. With this facility, the database programming language is able to develop distributed applications, e.g. distributed event-based systems.

6.2 Future enhancements

An ambitious improvement is implementing query processing optimization. To enhance performance, the inherent parallelism of distributed system may be exploited for query processing parallelism. Under some circumstances, change query execution order may improve performance greatly.

Given a query, there are generally a variety of methods for computing the answer. It is the responsibility of the system to transform the query entered by the user into an equivalent query that can be computed more efficiently.

Example

Suppose a user enters the following query,

```
>aldatp://mimi/R <- aldatp://mimi/T ijoin aldatp://mimi/S;
```

if the query optimization transforms the query into an equivalent query,

```
>aldatp://mimi/{ R <- T ijoin S};
```

Obviously, the new query is much more efficient (unless shipping query is more expensive than shipping three relations).

Example

Suppose a user enters the following query,

```
>R <- aldatp://mimi/(S ijoin T) ujoin aldatp://roo/(U ijoin V);
```

if aldatp://mimi/(S ijoin T) and aldatp://roo/(U ijoin V) can be executed in parallel, the parallel execution plan is apparently better than executing sequentially.

If the system supports replication or fragmentation transparency, it is the responsibility of the system optimizer to determine which fragments or replication need to be physically accessed in order to satisfy any given user request.

Example

If a relation R is in fragments R1 at "roo:`~tim/pubA`" , R2 at" roo:`~tim/pub`" and R3 at "mimi:~zwang26/pub", for the following querry

```
>T <- R ijoin S;
```

It needs to be replaced with

```
>T<- (aldatp://roo/~tim/pubA/R1 ujoin aldatp://roo/~tim/pub/R2
     ujoin aldatp://mimi/~zwang26/R3) ijoin  S;
```

However, it may be inefficient to rebuild the whole relation from its fragments. Take the example illustrated in figure 1.1. An *employee* relation is horizontally fragmented into *Employee1* and *Employee2* at different sites. *Employee* is the view taking unoin of *Employee1* and *Employee2*. If the user's query is

```
Ans <- [ id, name] where CS="department" and salary=5000 in Employee;
```

If we reconstruct *Employee* prior to pocessing, we obtain

```
Ans <- [ id, name] where CS="department" and salary=5000 in (Employee1 unoin Employee2);
```

However we could answer the query by only using fragment *Employee1*

```
Ans <- [ id, name] where CS="department" and salary=5000 in Employee1;
```

An important issue of fragmentation transparency is finding a query processing strategy based on the fragments rather than the relations, even though the queries are specified on the latter.

Semijoin is another basic technique for optimizing a sequence of distributed join operation [BC81, BG81b, KYY82]. It aims to remove tuples of a relation that fail to contribute to the result before shipping that relation. The main value of the semijoin is to reduce the size of the join operands and thus the communication cost. This is desirable particularly if network costs are high. However they might increase local costs.

A JRelix querry processor with an optimizer will transform the initial parser tree into an equivalent tree that is expected to require less-time to execute. By selecting an order of execution for these operators an execution plan is generated and then is executed.

Apart from optimizer, a coordinator or primary site is desirable to be added under some situations. In our implementation, all the sites are totally autonomous.

Although each physical computer has a root server which knows all the sites on the same physical machine, the global information about all the hosts linked by the Internet is unknown. A coordinator or primary site having global information is desirable for some applications.

For instance, a simple approach to support replication is keeping a **replica table** at each site. If a data item is replicated, the system must consult the **replica table** to choose a replica. If a coordinator exists, the coordinator can collect system table ".rel" from all the sites. So the coordinator not only knows the location of each database but can also get the relation names in each database from those ".rel" files. Then the coordinator is able to construct a relation or table called "Replica" having two attributes "location" and "relationName". Subsequently, it broadcasts the "Replica" to each sites. We suppose each relation has a unique global name or alias(view) for this method. Otherwise, we can use another solution. Whenever a relation is to be replicated, the relation name and the names of those sites keeping a replication are sent to the coordinator. Therefore, the coordinator is able to construct the replica table.

In addition, for future distributed transaction management implmentation, a coordinator or primary site may play a important role.

In short, this implementation is a modest work. It adopts a simple and efficient method to integrate distributed data processing and Internet capability into a database programming language. There still remains a lot of ground for further implementation to make the JRelix system more capable and powerful for distributed data processing and Internet applications.

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrision. Ps-algol: A language for persistent programming. In *In 10th Austrian National Computer conference*, pages 70–79, 1983.

[ABG84] R. Attar, P. A. Berstein, and N. Goodman. Site initialization, recovery, and backup in adistributed database systems. *IEEE Transaction on Software Engineering*, SE-10(6), 1984.

[Bak98] Patrick Baker. Java implementation of computations in a database programming language. Master's thesis, McGill University, Montreal, 1998.

[BC81] P. A. Bernstein and D. W. Chiu. Using semijoin to solve relational queries. *Journal of the ACM*, 28(1):25–40, 1981.

[BG81a] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM computing Surveys*, 13(2), 1981.

[BG81b] P. A. Bernstein and N. Goodman. The power of natural semijoin. *SIAM Journal of computing*, 10(4):751–771, 1981.

[BG82] P. A. Bernstein and N. Goodman. A sophisticate's introduceion to distributed database concurrency control. In *Proceedings of the International Conference on Very Large Data Bases*, 1982.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BHL83] E. Bertino, L. M. Hass, and B. G. Lindsay. View management in distributed database systems. In *Proceedings of the 9th International Conference on Very Large Data Bases*, October 1983.

[BL94] T. Berners-Lee. Universal Resource Identifiers in WWW. *RFC 1630*, 1994.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. *RFC 1945*, 1996.

[BLMM94] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). *RFC 1738*, 1994.

[BN97] P. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann, 1997.

[BRGP78] P. A. Bernstein, J. B. Rothnie, N. Goodman, and C. A. Papadimitriou. The concurrency control mechanism of SDD-1:a system for distributed databases. *IEEE-TSE*, SE(4:3), 1978.

[BRJS78] Philip A. Bernstein, James B. Rothnie, Jr., and David W. Shipman. Tutorial:distributed data base management. In *IEEE Computer Society*, 1978.

[BS80] P. A. Bernstein and D. W. Shipman. The correctness of concurrency control mechanism in a system for distributed databases (SDD-1). *ACM-TODS*, 5(1), 1980.

[BSR80] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie. Concurrency control in a system for distributed databases (SDD-1). *ACM-TODS*, 5(1), 1980.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design, Third edition.* Addison-Wesley, 2001.

[Cha83] A. Chan. Overview of an ADA compatible distributed databse manager. *ACM SIGMOD,San Jose,CA*, 1983.

[Che01] Yuling Chen. A G.I.S. editor for a database programming language. Master's thesis, McGill University, Montreal, 2001.

[CHM83] K. M. Chandy, L. M. Haas, and J. Misra. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2), 1983.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 1970.

[CP83] Stefano Ceri and Giuseppe Pelagatti. Correctness of query execution strategies in distributed databases. *ACM Transactions on Database Systems*, 8(4):577–607, 1983.

[CP84] Stefano Ceri and Giuseppe Pelagatti. *Distributed Database: Principles and Systems*. McGraw-Hill, 1984.

[Dat00] C. J. Date. *An Introduction to Database Systems, Seventh Edition*. Addison-Wesley, 2000.

[DJ84] Birrell A. D. and Nelson B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[DW80] C. Devor and J. Weeldreyer. DDTS: A testbed for distributed databse research. *Honeywell Report HR-80-268*, 1980.

[ES80] R. Epstein and M. R. Stonebraker. Analysis of distributed databse processing strategies. In *Proceedings of the International Conference on Very Large Data Bases*, pages 92–110, 1980.

[ESW78a] R. Epstein, M. R. Stonebraker, and E. Wong. Distributed query processing in a relational databse system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 169–180, 1978.

[ESW78b] Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational data base system. In *Proceedings of the ACM SIGMOD International Conference*, pages 169–180, 1978.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. *RFC 2616*, 1999.

[Fie95] R. Fielding. Relative uniform resource locators. *RFC 1808*, 1995.

[GBW+81] N. Goodman, P. A. Bernstein, E. Wong, C. L. Reeve, and J. B. Rothine. Query processing in SDD-1: A system for distributed databases. *ACM-TODS*, 6(4), 1981.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. San Mateo Calif: Morgan Kaufman, 1993.

[Gra78] J. Gray. *Notes on Data Base operating Systems*. in R. Bayer, R. M. Graham, and G. Seegmuller(eds), Operating Systems: An advanced course, New York : Springer Verlag, 1978.

[Gra79] J. N. Gray. A disscusion of distributed systems. In *Proc. congresso AICA 79*, October 1979.

[Gro96] Object Management Group. The common object request broker: Architecture and specification, version 2.0. OMG web site at:http://www.omg.org, 1996.

[Hao98] Biao Hao. Implementation of the nested relational algebra in Java. Master's thesis, McGill University, Montreal, 1998.

[He97] Hongbo He. Implementation of nested relations in a database programming language. Master's thesis, McGill University, Montreal, 1997.

[HS80] M. Hammer and D. Shipman. Reliability mechanism for SDD-1. *ACM-TODS*, 5(4), 1980.

[HY79] A. R. Hevner and S. B. Yao. Query processing in distributed databse systems. *IEEE Transactions on Software Engineering*, SE-5(3):177–187, 1979.

[Kna87] E. Knapp. Deadlock detection in distributed database. *ACM Computing Surveys*, 19(4), 1987.

[Koh81] W. H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM computing Surveys*, 13(2), 1981.

[KYY82] Y. Kambayashi, M. YoshiKawa, and S. Yajima. Query processing for distributed database using generalized semi-joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 151–160, 1982.

[Lin80] B. G. Lindsay. Site autonomy issues in R*: A distributed database management systems. *IBM Research Report RJ2927(36822)*, 1980.

[Lin81] B. G. Lindsay. Object naming and catalog management for a distributed database manager. In *Proc. 2nd Int. Conf. on Distributed Computing Systems*, 1981.

[Lin83] B. G. Lindsay. Computation and communication in R*: A distributed databse manager. In *Proc. 9th ACM Symp. on Operating Systems Principles*, 1983.

[Lit82] W. Litwin. Sirius systems for distributed data management. *Distributed Database, H.J. Schneider, ed. North-Holland*, 1982.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore dayabase system. *Communications of the ACM*, 34(10):50–63, 1991.

[LS76] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. *Technical Report, Computer Science Laboratory, Xerox*, 1976.

[Mer84] T. H. Merrett. *Relational Information Systems*. Reston Publishing Co., Reston, VA, 1984.

[Mer01] T. H. Merrett. Attribute metadata for relational OLAP and data mining. In *Proceedings, Eighth Biennial Workshop on Data Bases and Programming Languages*, pages 65–76, Monteporzio Catone, Roma, Italy, Sept.2001.

[Mer02] T. H. Merrett. Database programming meets Internet programming. Technical report, School of Computer Science, McGill University, 2002.

[ML83] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the @nd ACM AIGACT-SIGOPS Symposium on the Principles of Distributed Computing*, 1983.

[Mor88] R. Morrison. Ps-algol reference manual. *Technical Report 12, University of St. Andrews*, 1988.

[NW82] E. J. Neuhold and B. Walter. An overview of the architecture of the distributed data base system porel. *Distributed Database, H.J. Schneider, ed. North-Holland*, 1982.

[Ora02a] Oracle. Oracle9i advanced replication release 2 (9.2) part number a96567-01. Oracle web site at: http://www.csis.gvsu.edu/GeneralInfo/Oracle/server.920/a96567/toc.htm, 2002.

[Ora02b] Oracle. Oracle9i database administrator's guide release 2 (9.2) part number a96521-01. Oracle web site at: http://www.csis.gvsu.edu/GeneralInfo/Oracle/server.920/a96521/title.htm, 2002.

[OV99] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd edition.* Prentice-Hall, 1999.

[Rao95] B. R. Rao. Making the most of middleware. *Data Communications International*, 24(12):89–96, 1995.

[RG77] J. B. Rothnie and N. Goodman. An overview of the preliminary design of SDD-1: A system for distributed database. In *Proc. 2nd Berkeley Workshop on Distr. Data Manag. and Computer Networks*, 1977.

[RJG77] James B. Rothnie, Jr., and N. Goodman. A survey of research and development in distributed database management. In *Proceedings of the 3rd International Conference on Very Large Data Bases*, pages 48–62, Tokyo,Japan, October 1977.

[Rot80] J. B. Rothnie. Introduction to a system for distributed databases (SDD-1),. *ACM Trans. on Database Systems*, 5(1):1–17, 1980.

[RSL78] D. J. Rosenkrantz, R. E. Stearns, and P.M. Lewis. System level concurrency control for distributed database systems. *ACM Transactions on Database systems*, 3(2), 1978.

[Sch77] Joachmim W. Schmidt. Some high level language constructs for data of type relation. *ACM transactions on Database Systems*, 2(3):247–261, 1977.

[SDV96] Sriram Sankar, Rob Duncan, and Sreenivasa Viswanadha. Java Compiler Compiler (JavaCC)-The Java Parser Generator. JavaCC web site at: http://www.suntest.com/JavaCC/, 1996. The web site contains documentations, FAQs, newsgroups, and softwares for JavaCC and JJTree.

[SH] Stonebraker and Hellerstein. Distributed DBMS: Overview and concurrency control. *Readings in Database Systems, 3rd Edition.*

[Ske81] D. Skeen. Non-blocking commit protocols. In *Proceedings of the ACM SIGMON International conference on the Management of Data*, 1981.

[SKS97] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Third Edition.* McGraw-Hill, 1997.

[SM95] R. Srinivasan and Sun Microsystems. RPC: Remote procedure call protocol specification version 2. *RFC 1831*, 1995.

[Smi82] J. M. Smith. Multibase: Integrating heterogeneous distributed database systems. *Proc. National Computer Conference*, 1982.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1 , The Protocols.* Addison-Wesley, 1994.

[Ste96] W. Richard Stevens. *TCP/IP Illustrated, Volume 3 , TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols.* Addison-Wesley, 1996.

[Sto86a] M. Stonebraker. *The Design and Implementation of Distributed INGRES, in The INGRES Papers, M. Stonebraker (ed.).* Addison-Wesley, Reading, MA, 1986.

[Sto86b] Michael Stonebraker. *The INGRES Papers: Anatomy of a Relational Database System.* Addison-Wesley, 1986.

[Sun00] Weizhong Sun. Updates and events in a nested relation programming language. Master's thesis, McGill University, Montreal, 2000.

[SWKH76] M. R. Stonebraker, E. Wong, P. Kreps, and G. D. Held. The design and implementation of ingred. *ACM Transactions on Database Systems*, 1(3), 1976.

[Tib95] Fred Tibbets. Corba: A common touch for distributed applications. *Data Comm Magazine*, 24(7):71–75, 1995.

[Wil81] R. Williams. R*: An overview of the architecture. *IBM Research Report RJ3325*, 1981.

[Won77] E. Wong. Retrieving diepersed data from SDD-1: A system for distributed database. In *Proceedings of the Berkely Workshop on Distributed Data Management and Computer Networks*, 1977.

[Won83] E. Wong. Dynamic rematerialization-processing distributed queries using redundant data. *IEEE transactions on Software Engineering*, SE-9(3):228–232, 1983.

[WW] Wollrath and Jim Waldo. Trail: Rmi. Java RMI web site at: http://java.sun.com/docs/books/tutorial/rmi/index.html. The web site contains documentations and examples.

[Yua98] Zhongxia Yuan. Java implementation of the nested domain algebra in a database programming language. Master's thesis, McGill University, Montreal, 1998.

[Zhe02] Yi Zheng. Abstract data types and extended domain operations in a nested relational algebra. Master's thesis, McGill University, Montreal, 2002.