A Java Implementation of a Linda-like Tuplespace System with Nested Transactions

A thesis presented in partial fulfilment of the

requirements for the degree of

Master of Science

in

Computer Science

at Massey University, Albany, New Zealand

Yinan Yao

2006

Abstract

The *Tuplespace* model is considered a powerful option for the design and implementation of loosely coupled distributed systems. In this report, the features of the Tuplespace model are examined as well as the issues involved in implementing such a Tuplespace system based on Java. The system presented includes the function of *Transactions*: a collection of operations that either all succeed or all fail. The system also permits *Nested Transactions*: an extension of transactions. Nested transactions have a multi-level grouping structure: each nested transaction consists of zero or more operations and possibly some nested transactions. The key advantages offered by nested transactions include that they enable the failure of an operation to be isolated within a certain scope without necessarily aborting the entire transaction, and they allow programmers to sub-divide a complex operation into a number of smaller and simpler concurrent operations. The other features of nested transactions are also examined in this report. Finally, the testing results indicate that it is possible to build an efficient, scalable, and transaction secured distributed application that relies on the Tuplespace model and the system developed for this research.

Acknowledgements

First, thanks to my research supervisor Heath James who has aided me greatly throughout my 2 years of postgraduate study. I know I couldn't finish my thesis without his excellent guidance and support. Thanks to many other lecturers in the Computer Science department who taught me a lot in my first year study at Massey University.

Next, the research facilities provided by Massey University were great. I was given 24-hour access to the computer laboratory. The university library contains a large number of useful materials. And also, I have to thank Massey University for the Masterate Scholarship that was a great financial support.

Last, I have to thank my parents who have offered me constant support and encouragement. Their support was always the key reason that kept me going.

Table of Contents

Table of Figures

Table of Code Samples

1. Introduction

Distributed computing systems are being built and used more and more frequently in today's computer software development industry because they offer a number of extra advantages over traditional centralized programs for certain sets of complex problems, including performance improvements, enhanced scalability, resource sharing, fault tolerance, load balancing, and system design elegance. However, distributed computation also requires a programming model that accommodates the particular problems that occur.

The familiar programming models that are widely adopted by distributed computing as described in [1] are:

- Remote Procedure Calls/ Remote Method Invocations/ Common Object Request Broker Architecture (RPC/RMI[2]/CORBA[3])
- Messaging/Message Passing Interfaces (MOM/MPI)
- Interacting peers (P2P)

A fourth model could be called *Tuplespace* or *shared objectspace*, which will be presented in this thesis.

The goal of this research is to design, build and test a shared objectspace system named "*Tuplespace*" for dynamic communication, coordination, and data- and object-sharing. This Tuplespace system is aimed to allow participants in a distributed solution to easily exchange tasks, requests and information, and gives developers the ability to create and store objects with persistence under single/multiple level transaction controlled manner.

Because computer network hardware and underlying protocols (e.g. Sockets, URL, TCP/IP, UDP, and Data packet switching, Network hardware) are already fairly well understood, this thesis will not provide an examination of these issues. Rather, this thesis will be concerned mainly with the software aspects of achieving a reliable and transactionally secured Tuplespace system for computers that communicate via a network.

1.1 The Tuplespace Concept

The concept of a Tuplespace was first introduced by David Gelernter at Yale University in the mid-1980's[4]. Gelernter developed a coordination language for concurrent programming named *Linda* which provides a communication mechanism based on a logically shared memory space called *tuple space*. Figure1 shows the Linda model that facilitates process communications through the exploitation of the concept of a Tuple-space. By effectively combining a "persistent" data store with a small set of operations (i.e. `in()`, `out()`, `rd()`, `eval()`), Gelernter introduced the concept of loosely coupled process communication that could be independent of both space and time[5].

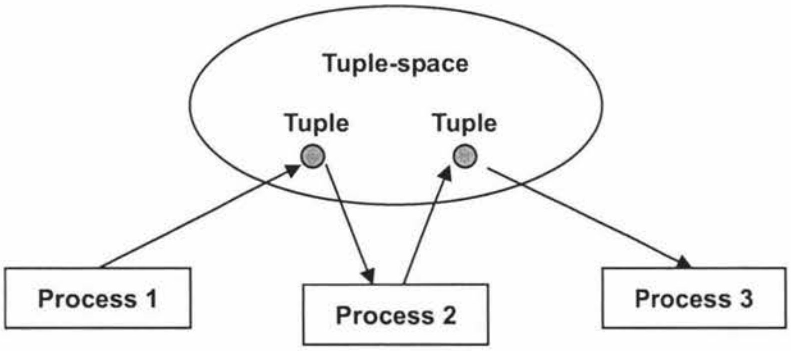


Figure1 – Tuple-space based process communication

On a shared memory multi-processor system, the Tuple-space is actually shared. However, on distributed memory systems (such as a network of workstations) the Tuple-space is usually distributed among the processing nodes. This thesis will focus on distributed memory systems as they are more commonly used in today's distributed systems. As illustrated in Figure2, a Tuple-space is a particular "pool" of objects where everyone can put and retrieve objects. Each Tuplespace takes care of the details of data transfer, persistence, communication, synchronization, transaction management, etc. For example, instead of focusing on the details and problems associated with distributed computing data transfer between components **A** and **B**, a programmer can simply have **A** place an object in a Tuple-space, and the Tuple-space will take care of that object until **B** asks for it. The details of how to achieve this will be discussed soon in this thesis. As today's Tuplespace models are initialized and strongly influenced by Linda systems, the Tuplespace system presented by this research is developed based on the concept of Linda systems and is similar to Linda systems in that they store collections of information for future computation, are driven by value-based lookup and include only a small set of easy to use operations.

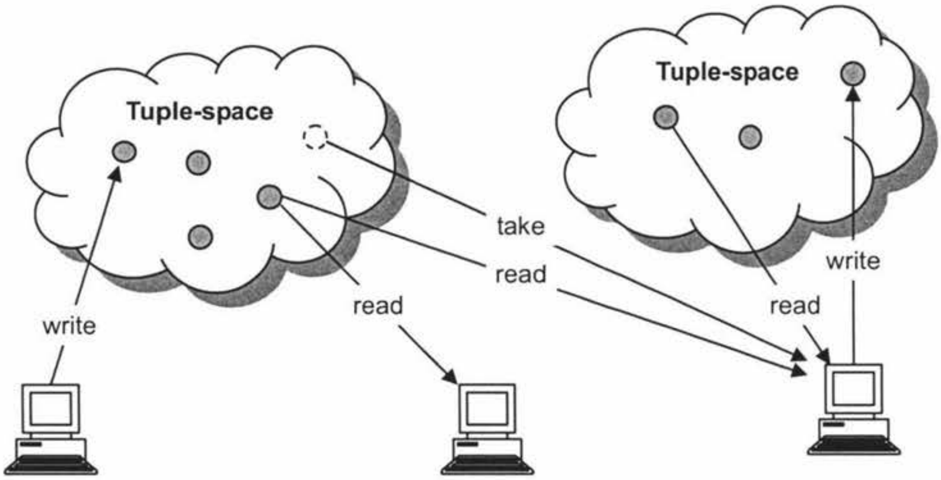


Figure 2 – Distributed nodes use spaces and simple operations to coordinate activities

1.2 Plan of the Report

Here is a brief review of the organization of the report. Chapter 2 presents an overview of the Tuplespace model and its principles. This chapter discusses the features and advantages of Tuplespace model over the other distributed computation models. Chapter 2 also includes a brief introduction and comparison of the major commercial Tuplespace products available in the market.

Chapter 3 provides a detailed technical exposition of how the proposed Tuplespace system is structured and implemented. It introduces the methods provided by the

Tuplespace system to client-side programmers. Chapter 4 explores the underlying components and protocols that form the Tuplespace system.

Chapter 5 discusses the idea of transactions in some detail, including the ACID properties and algorithms used in this Tuplespace system for locking, state restoration, transactional operation and commit/abort processes. Chapter 6 introduces the idea of nested transactions and explores it in detail. Chapter 6 is somewhat parallel to Chapter 5 in that it extends the locking, state restoration, and completion algorithms of Chapter 5 to nested transactions.

Chapter 7 builds on the techniques of Chapter 3, Chapter 4, Chapter 5, and Chapter 6, by developing some small programs to test the usability, functionality and the performance of the proposed Tuplespace system.

Chapter 8 attempts to offer some suggestions for further research directions. Chapter 9 presents a summary with some conclusions.

2. An Overview of Tuplespace Architecture

This chapter presents some fundamental concepts upon which the Java-implemented Tuplespace system is built in later chapters. It will describe the principles of Tuplespace systems and introduce some unique advantages of a Java-implemented Tuplespace system in comparison with the other distributed computing models. At the end of this chapter, it will briefly introduce some major commercial Tuplespace products available in the market.

2.1 Principles of Tuplespace Systems

As mentioned in Chapter1, a Tuple-space is a "pool" of objects where everyone can put and retrieve objects. In this report, the term *Tuple-space*(with a hyphen in the middle) refers to the actual pool that stores tuples, and the term *Tuplespace Server* refers to the Tuplespace server-side program that accesses the Tuple-space and provides services to the clients. As the name suggests, there is the notion of a *Tuple*, which is a container object holding a combination of user data and fields. When data objects are stored or retrieved, they are annotated with a series of fields and passed between the Tuple-space and clients.

In order to retrieve data from the Tuple-space, the client presents to the Tuple-space a template (also a tuple object) which becomes the parameter for a "matching" process running on the Tuplespace server side. A tuple in the Tuple-space that "matches" this template will be returned to the client.

The retrieval process can be a consuming process: when a matching tuple is found, it may be removed from the Tuple-space. The retrieval process can also be a non-consuming process when no matching tuples are found or matching tuples are just copied to the client. The retrieval operation can be a blocking operation with or without a client-specified timeout value (i.e. how long the clients is willing to wait for response to be returned from the Tuplespace server) or non-blocking (returns immediately no matter if a matching tuple is not found).

The properties of a Tuplespace system can be summarized as the following facts:

- **shared** A Tuple-space is a shared storage where everyone shares the same information at the same time. Multiple Tuple-spaces can exist on the same network. The Tuple-spaces residing on different processes/nodes can also interact with each other to share information.
- **persistent** When an object is placed in the Tuple-space, it stays there until it is explicitly removed or the Tuplespace server is shut down. The existence of the data objects does not rely on the existence of the process that created it, in fact, a tuple can exist independently of any clients[5].
- **associative** Clients don't need to know the name or the location of an object in a Tuple-space in order to find it. Instead, clients create a template object which is also a kind of tuple object and it is made up of values/properties (fields) they're seeking, and clients receive objects that match the values/properties defined in the template from the Tuple-space. This type of associative lookup gives the programmer great flexibility and true "loose coupling" when programming systems. Programmers can program without the need of knowing names or locations of other components of their application.

2.2 Advantages of Tuplespaces Implemented in Java

As mentioned in Chapter 1, several solutions to the problems of distributed computing have been proposed and implemented over the years: RPCs, MPI, PVM, CORBA, RMI, etc. However, many problems and inefficiencies exist within these technologies. For example, a general problem with RPC-type solutions is that client and server programs must know a great deal about each other in order to work. In other words, they are tightly coupled. If there is a change in a client, a relative change must be made in the server. A great deal of coordination is required and any single change in one program can require a number of rewrites in the same program and other dependent programs.

In contrast, Java implemented Linda-like Tuplespaces are designed to provide a simpler, more powerful and more elegant model for dynamic communication, coordination, and data- and object-sharing [5], [6], [7]. A number of extensions to the original Linda concept have been proposed[8], [9], [10], [11], [12], [13]. For example, an *Object Space*[9], that contains objects and message and introduces *object orientation* to Tuplespaces. In [9], each object encapsulates a state in form of multiset of tuples and methods in form of rewriting rules. [10] suggests the use of *semantic* templates in XML format that match tuples structurally and gives the processes connected to the XML Tuplespace the freedom to exchange and transfer any information. This research focuses on implementing a Linda-like Tuplespace system that is simple to use and contains single-level/multiple-level transactional controls. The major advantages of a Java-based Tuplespace implementation are described as below:

- Java-implemented Tuplespaces run in the Java Virtual Machine (JVM). Since JVMs exist for most of today's platforms, this means that such system can be used by all those platforms, too. The Java-implemented Tuplespace users don't have to worry about cross-platform differences because the system exists in networked JVMs.
- Java-implemented Tuplespace is simple to use. There are only a few additional simple methods to learn. Users don't have to worry about multiple clients, synchronization, data transfer, persistence, transactions, etc., because those details are handled by the Tuplespace service itself.
- Operations are transactionally secure. A well designed Tuplespace system needs to allow a series of operations to be done under transactional controls to guarantee that operations are atomic (that is, either all of the operations are applied, or none of them is). Transactions are not only important to many

database and network applications, but also offer a method to handle partial failure by making use of Nested Transactions.

- Java-implemented Tuplespace systems allow exchanging executable objects. As data stored in the Tuple-space are Java objects, clients can easily read/take objects from the Tuple-space, create a local copy, and access its public fields, invoke its methods.
- Traditional Linda systems have not used rich typing. Java-implemented Tuplespace systems take strong typing mechanisms from the Java platform type-safe environment. In the presented Tuplespace system, tuple types are of specific different classes for the Java platform, and templates for one type would never match another, even if the fields contained are identical.
- A Tuple-space can also be arbitrarily large, spanning multiple servers – this means that if one Tuplespace server fails, the Tuplespace server can transfer the entire space to another server and continue operation. And also, if there is no matching tuple found in one Tuple-space, the client's request can be transparently redirected to other Tuple-spaces. Thus, Tuplespace system can overcome some of the latency and partial failure problems that face distributed computing.
- Java-implemented Tuplespace systems can be used to balance work load among client computers. For example, in the Master-Worker model [14], [15], the master process generates tasks that need be processed and stores them as objects in Tuple-space(s), the worker processes retrieve tasks from the spaces whenever they have available resources and the results are written back to the space for later computation.
- The Tuplespace functionality is very similar to a database, but it is not. Relational databases understand the data they store and manipulate it directly via SQL languages [16]. There are no general queries in the presented Tuplespace system, only "exact match" or "don't care" for a given field. In this Tuplespace system, programmers have an option to design their own matching policy for fields which would permit them a great deal of flexibility.

2.3 Current Major Tuplespace Implementations

As the Linda Tuplespace concept has been around for many years, there are a number of commercial Tuplespace products available in today's market. Tuplespace implementations in Java are mainly found in products like IBM's TSpaces, Sun's JavaSpaces and GigaSpaces. All of these products are developed based on Linda concept and written in Java and share a lot of similarities, but they each has unique features and is also competitor of each other. The Tuplespace system developed in this research has borrowed some ideas from these existing products, especially from Sun's JavaSpaces, so it is worth presenting some discussions about these products.

IBM TSpaces

IBM has created a version of tuple-spaces, called TSpaces[17]. Like the other major products, it is written in Java, and offers a small set of methods as well as an object store. But TSpaces offers different capabilities: it has a built-in database with transaction and indexing support as well as a simplified query language. The addition of database services provided by TSpaces is a significant feature and advantage over Sun's JavaSpaces. IBM describes TSpaces as "the common platform on which we build links to all system and application services"[17]. The implementation of TSpaces is simple. All that is required is that a single server process be running on the network. Applications wishing to make use of the TSpaces service need only know the network hostname of the computer running the server. TSpaces provides a large number of operations over the basic Linda operations, such as `delete`, `deleteAll`, `multiWrite`, `multiUpdate`.

Sun JavaSpaces

Sun JavaSpaces is both an application program interface and a distributed programming model[16]. It allows a network of computers to co-operate and offer space operations, distributed events, leases, and transactions. JavaSpaces is a complex product and relies heavily on a number of other technologies from Sun.

JavaSpaces supports the basic Linda operations, including `write`, `take`, `read`, `notifyAll`. JavaSpaces forms part of the Sun Jini system, and so makes extensive use of the other technologies of Jini. Network support is provided by the Java RMI protocol. Furthermore, distribution of classes to clients is handled by the standard Internet hypertext protocol (HTTP). This means that before a JavaSpaces application can be started the following set of services must be running:

- a web (HTTP) server
- an RMI activation server (part of the standard RMI software bundled with Java)
- a Jini lookup service
- a Jini transaction manager
- a JavaSpaces server

Most of these services also require extensive work to set up, further adding to the overall complexity of using JavaSpaces. Applications are also required to run a security manager, whether security checking is required or not. This is actually a major disadvantage of JavaSpaces mechanism as it is complex to set up.

GigaSpaces

GigaSpace is a relatively recent system developed as a commercial implementation of the JavaSpaces specification[18]. As such it is compliant with the Sun specifications, while adding a number of new features. These include operations on multiple tuples, updating, deleting and counting tuples, etc.

2.4 Uses of Tuplespace Architectures

Comparing with all other currently fashionable approaches, Linda-like Tuplespace model and implementation have been proved a simpler and more useful approach for distributed computing in many situations. The distinct features and benefits of Linda-like Tuplespace architecture result its use growing steadily[5] in both industry and research fields. This section briefly introduces a few examples of how Tuplespace concept could be used in solving real problems:

- Mojave[19] is a project that implements auto-configuring services using reactive and mobile agents to dynamically deploy distributed services over a complex network in an optimal manner. Mojave uses Tuplespace as the messaging and mobility hub to enable changes in an application's coordination behaviour without rewriting components. Mojave's use of Tuplespaces allows for more powerful message routing than is possible with RMI, because Tuplespaces support message buffering as well as database like matching, both of which are enablers for smart routing[19].
- Digital Equipment Corp. plans to demonstrate a distributed parallel computing application built with PAX-1 network development toolset. PAX-1, an implementation facilitates parallel processing with Tuplespace and offers full peer-to-peer distributed processing[20]. The DEC demonstration will show an application that creates a 'network supercomputer' by combining the resources of several workstations. *Sandia National Laboratories* also implemented a distributed parallel processing system based on the Linda programming[21]. SNL's system allows a single application program to utilize many machines on the network simultaneously and achieved performances considerably faster than that of a Cray-1s. Several collections of machines have been used including up to eleven DEC VAXes, three Sun/3 workstations, and a PC.
- Semantic Web Spaces[22], a middleware platform for real world Semantic Web application. By applying a Tuplespace-based approach to the concurrent interaction of multiple clients with distributed knowledge repositories, a simple, yet powerful coordination model in which parallel and distributed Semantic Web[23] processes can be uncoupled in space and time can be obtained[22].

Since the concept and advantages of Java implemented Tuplespace systems are relatively easy to understand, and there is only a small, easily mastered set of methods, the Java implemented Tuplespace systems (i.e. JavaSpaces, TSpaces) should definitely be considered as an option when developing distributed applications. In the following chapters, the details of designing and implementing a Tuplespace system in Java will be presented.

3. Design of the Java Implemented Tuplespace System

In this chapter, the design of the proposed Tuplespace system is explained. There are three primary operations that clients can invoke. Each operation needs a tuple object as a parameter, which is also named as template when it is used in retrieval operation. This chapter describes how clients can connect to the Tuplespace server and the essential components building up the Tuplespace system, including tuples, templates, tuple matching principles and the details of the three primary operations, which are:

- `out(ITuple tuple, TransactionManager.TransactionProxy trx):` Write the given tuple into the Tuple-space.
- `rd(ITuple template, TransactionManager.TransactionProxy trx, long timeout):` Read a tuple from the Tuple-space that matches the given template.
- `tk(ITuple template, TransactionManager.TransactionProxy trx, long timeout):` Read a tuple from the Tuple-space that matches the given template, and remove it from the space.

3.1 Overview of the Tuplespace Model

The distributed system running the proposed Tuplespace system consists of a number of nodes (real/abstract computers) that communicate by sending *messages* over a communication network. Some nodes act as clients and consist of a processor and memory. Clients may leave (e.g. crash) and rejoin (e.g. recover) the system at any time, and new clients may join the system over time. Some other nodes are treated as servers and these nodes are supposed to be up all the time in order to provide persistent services and tuple storages. Clients connect to the Tuplespace servers, and write tuples to the Tuple-space held by the servers. And also, clients can send template to the Tuplespace servers, and the servers look for matching tuples in their Tuple-spaces and return results to the clients. Strictly speaking, this Tuplespace system is a server-centric model in which most of the processing takes place in the remote Tuplespace server (e.g. tuple matching, tuple storage). Providing that the server-centric Tuplespace system is implemented in a fashion that correctly handles synchronization issues, it can easily be shared among multiple clients. Each node in the system is located by IP address, which means that any node in the system needs to have an IP address. As long as the client node knows the IP address, the computer name or the URL address of the server node which holds the Tuplespace service, the client can connect to the server and start writing or retrieving tuples. Servers are also able to communicate with each other to share information, i.e. tuples. The communications between client/server and server/server are in duplex directions.

3.2 Connecting to a Tuple-space

To access the services provided by a Tuplespace server, the client first has to establish a connection with the Tuplespace server. The client first needs to create an object of `tuplespace.TupleSpace`, which is a proxy representing the Tuplespace service on the client side. This proxy is not a full implementation of the Tuplespace service. Instead, it links to the remote Tuplespace server that provides access to the desired methods and Tuple-space via use of message passing and remote method invocation. The client invokes the operation through the `TupleSpace` proxy object, and the `TupleSpace` proxy object handles all the underlying data formatting and data transferring between the client and the actual Tuplespace server and all of the underlying processes are kept invisible to the client. From the client's point of view, he is operating on the Tuple-space directly and locally. The `TupleSpace` constructor has 3 parameters: the conceptual name of the Tuplespace; the IP address, the computer name or the URL address of the computer on which the target Tuple-space resides; and the socket port on which the Tuplespace server is monitoring for clients' incoming requests. As there may be a number of Tuplespace servers running at the same time, the conceptual name of the Tuplespace server is used to distinguish the Tuplespace server from each other in a more understandable manner. Although Tuple-spaces can share data (tuple) with each other, the client's `out(...)` operation is only done to the Tuple-space that the client connects to explicitly (also called the primary Tuplespace).

The interactions among the cooperating Tuplespace servers are absolutely transparent to the client. As shown in Figure3, if Tuple-space "Auckland" doesn't have the tuple that client A is looking for, then it will automatically send the template to another Tuplespace server (i.e. alternative Tuplespace) to search for a matching tuple while client A has no idea about which Tuple-space actually contains the returned tuple. One client can connect to multiple Tuple-spaces at one time, but each connection has to be set up explicitly. As shown in Figure3, client A connects to Tuple-space "Auckland", and "Auckland" can interact with "New York", the tuples written by client A are stored in Tuple-space "Auckland". Client A can also set up another "direct" connection with "New York" and write tuples to "New York". The sample code for establishing connections with Tuple-spaces is shown as below:

```
TupleSpace ts, t2;

InetAddress serverAddress = null;
InetAddress serverAddress2 = null;

try {
    serverAddress = InetAddress.getByName("computer1");
    serverAddress2 = InetAddress.getByName("computer2");
}
catch(UnknownHostException ex)
{
    System.out.println(ex);
}

ts = new TupleSpace("Auckland" , serverAddress, 8880);
ts2 = new TupleSpace("NewYork" , serverAddress2, 8880);
```

Code Sample 1- Establish connection with Tuplespace Server

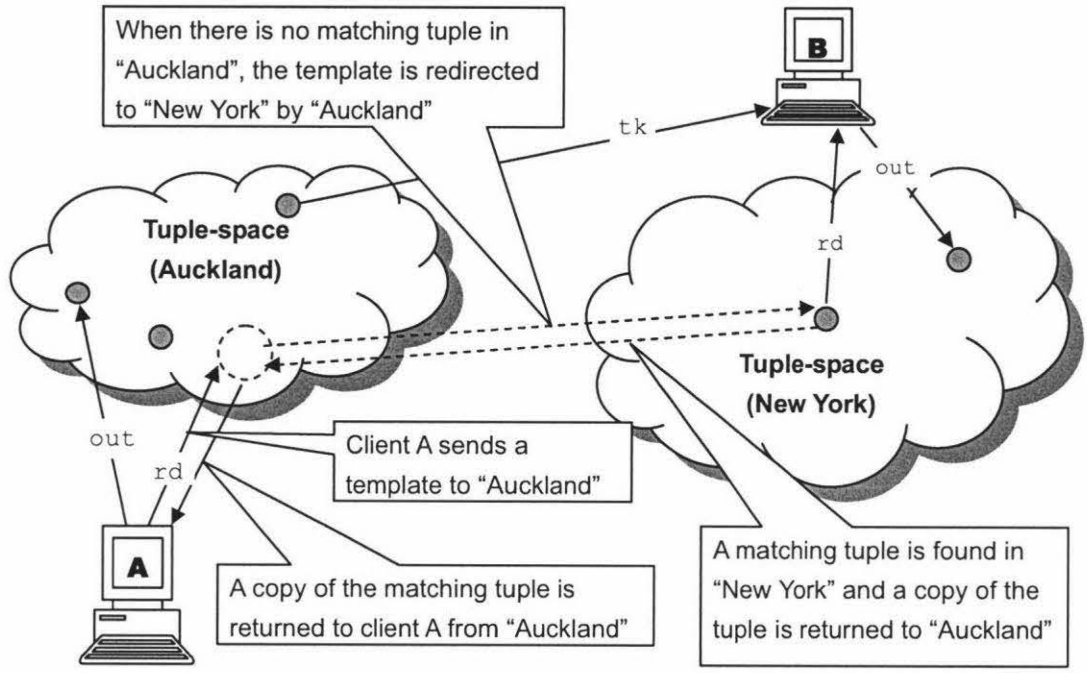

When there is no matching tuple in "Auckland", the template is redirected to "New York" by "Auckland"
B
tk
out
rd
Tuple-space (Auckland)
Tuple-space (New York)
out
rd
Client A sends a template to "Auckland"
A
A copy of the matching tuple is returned to client A from "Auckland"
A matching tuple is found in "New York" and a copy of the tuple is returned to "Auckland"

Figure3 – Client's request handled by cooperating Tuple-spaces

3.3 Tuples and Templates

Just connecting to the Tuplespace server is not enough for the client to access the Tuplespace service. A tuple/template object (In JavaSpaces, tuple/template is of type `Entry` in package `net.jini.core.entry`[24]) and is essential for any one of the three primary Tuplespace operations to work. Both tuple and template are represented by objects of the class `tuplespace.Tuple`, which contains an ordered set of objects (also called `Fields`). Any field object has to be of the class `tuplespace.Field` or its subclasses and implements `java.io.Serializable` interface. The code for creating and defining a tuple is demonstrated in Code Sample2.

```
Tuple template = new Tuple();
template.add(new Field().setValue(new String("Test")));
template.add(new Field().setValue(new Integer(10)));
template.add(new Field().setType(SomeClass.getClass());
```

CodeSample2 – Creating tuples and templates

A tuple object can have as many fields as necessary and the programmer can create its own tuple type by inheriting from the `tuplespace.Tuple` class. The fields contained in a tuple are used for associative matching (The matching principles used in the proposed Tuplespace system will be discussed in detail in Section 3.5 Matching Tuples) and can also be used to represent usable data. The tuple object can also have other attributes rather than fields to carry useful data, as shown below.

```
class FlightTuple extends Tuple implements Serializable{
   public String airline;
   public String departPort;
   public String destPort;
   public Date departDate;
   public Integer seatAvailable;
   public Time departTime;
   public Time landTime;
   ...... }
```

Normally, one would do two things with tuples: Store them in Tuple-space and retrieve value of the field or use them as template for associative matching.

Tuples are transmitted as raw bytes across the network using Java standard serialization mechanism. Multiple copies of the same tuple can be written to the Tuple-space. On the server side, the raw bytes are deserialized back to an identical tuple object for storing or retrieval.

3.4 Storing Tuples in the Tuple-space

The tuple is stored into the specified Tuple-space through the `out(...)` operation. The `out(ITuple tuple, TransactionManager.TransactionProxy trx)` operation takes two parameters: the tuple object to be written; a `TransactionProxy` object which represents the transaction under which the `out(...)` operation is done. Details of transactions will be discussed in Chapter 5.

The tuple passed to the `out(...)` is not affected or modified by the operation. Each `out(...)` operation places a new tuple into the specified space under the specified transaction, even if the same tuple object has been used in more than one `out(...)` operation.

If `out(...)` returns without throwing an exception, that tuple is committed to the space, possibly within a transaction (see Chapter 5 Transaction). If any exception is thrown, the tuple was not written into the space. The code sample for writing tuples to a Tuple-space is shown as below:

```
......
Tuple t1, t2;

t1 = new Tuple();
t1.add(new Field().setValue(new String("Test")));
t1.add(new Field().setValue(new Integer(10)));

t2 = new Tuple();
t2.add(new Field().setValue(new String("Test2")));
t2.add(new Field().setValue(new Integer(12)));

try {
    ts.out(t1, null);
    ts.out(t2, null);
}catch(TransactionException e) {
    System.out.println(e);
}catch (TupleSpaceException e) {
    System.out.println(e);}
```

Code Sample3 – Define and write tuples to Tuple-space

`TransactionException` will be thrown if exception happened due to any sort of transaction errors, for example the transaction specified doesn't exist on the server or it has already completed.

`TupleSpaceException` will be thrown for any exceptions except transaction related exceptions, for examples, when a tuple can not be deserialized probably by the server, a `TupleSpaceException` exception will be thrown.

The consequence of the `out(...)` operation is that the tuple is serialized into raw bytes and packaged accordingly (i.e. controlled by certain protocols. The details of the protocols are discussed in Chapter 4) and passed to the Tuplespace server over TCP/IP connection. When the data package arrives at the server, the raw bytes are then deserialized back to a tuple object along with some additional information and stored into the space. This Tuplespace system is built based on the assumption that the data packages arrive at the server accurately (The assumption will be further explained in Chapter 4).

3.5 Matching Tuples

The most basic operation of a Tuplespace server is matching tuples. In order to match, the client first has to create a template which defines his matching request. Templates are in the same form of Tuples used for retrieval purposes. During retrieval process, a template is used for the selection of tuples that can be retrieved: all the tuples "matching" the template are candidates for retrieval. Some of the fields in the template can be "wildcards". Wildcards do not contain values or types, so they match any value of any type. Tuples and templates are ordered sets of fields. The fields are objects of type `tuple.Field`; each field object holds a value and the type of the value. The *Nth* field of a template will only match the *Nth* field in the tuples. Templates will never match a tuple with a different number of fields.

The matching process is typesafe in the sense that the matching checks both the type and the value of the fields. Therefore, an `Integer` field and a `Long` field will never match, regardless if they have the same number value. Unlike Sun's JavaSpaces, this Tuplespace system implements strict type checking, which means that a superclass object doesn't match its subclasses objects, i.e. a template with a `Number` field will not match fields of type `Long` or `Integer` even they are subclasses of `Number`. And also, a template of class `DerivedTuple` will not match tuples of class `Tuple` even they have the same fields. In Sun JavaSpaces, it allows matching of subtypes – a template match can return a type that is a subtype of the template type.

The use of wildcards can be typesafe as well, i.e. it will match any value of a field with a specified type. For example, a field defined as `tuple1.add(new Field().setType(String.class))` is a wildcard that match any fields with a `String` type regardless of the actual content.

Consider the tuples and templates in the box below:

```
Tuple t1 = new Tuple();
t1.add(new Field().setValue(new String("Test")));
t1.add(new Field().setValue(new Integer(10)));
t1.add(new Field().setValue(new SomeClass(“Some Class”)));

Tuple t2 = new Tuple();
t2.add(new Field().setValue(new String("Test")));
t2.add(new Field().setValue(new Integer(12)));
t2.add(new Field().setType(SomeClass.class));

Tuple template1 = new Tuple();
template1.add(new Field().setValue(new String("Test")) );
template1.add(new Field().setValue(new Integer(10)) );
template1.add(new Field().setValue(new SomeClass(“Some Class”)));

Tuple template2 = new Tuple();
template2.add(new Field().setType(String.class));
template2.add(new Field().setType(Number.class));
template2.add(new Field().setType(SomeClass.class));

Tuple template3 = new Tuple();
template3.add(new Field().setType(String.class));
template3.add(new Field().setType(Integer.class));
template3.add(null);
```

The different templates will match tuples in the following manner:

1. `template1` matches `t1`, each field matches on type and value.
2. `template2` matches no tuple, because the system performs strict type checking. Although `Number` is super class of `Integer` class, they don't match in this system.
3. `template3` matches `t1` and `t2`. Field 1 matches on type, field 2 matches on type, and the last field is of value "`null`" so it is typeless and matches any field.

3.6 Partitioning the Tuple-space

For most of the applications that will use the Tuplespace service, there will be many tuples to match and each Tuple-space may store a large amount of tuples. Even with a simple and efficient algorithm of tuple matching, sequentially matching through all tuples in the Tuple-space would be a huge waste of time and largely defeat the purpose of distributing programming in the first place. Thus it is desirable to divide the Tuple-space into partitions to minimize the number of tuples examined in any match.

As the present Tuplespace system implements strict type checking when matching tuples, it is natural to divide the Tuple-space into partitions (subspaces) based on the tuple types. The tuples of different types are stored in different "subspaces" as shown in Figure4. Therefore, the Tuplespace server will firstly match the ***type*** of the template against the type of the subspaces, if there is no subspace of this template class, the Tuplespace server can immediately decide that there is no matching tuple existing in this Tuple-space without comparing against all the tuples. If the subspace for that class exists in the space, then the server only needs to search inside that particular subspace. When a tuple is written to the space, the Tuplespace server checks the type of the tuple and adds it to the associated subspace. If the tuple is of a new type which doesn't exist in the Tuple-space yet, and the Tuplespace server will create a new subspace for the new type.

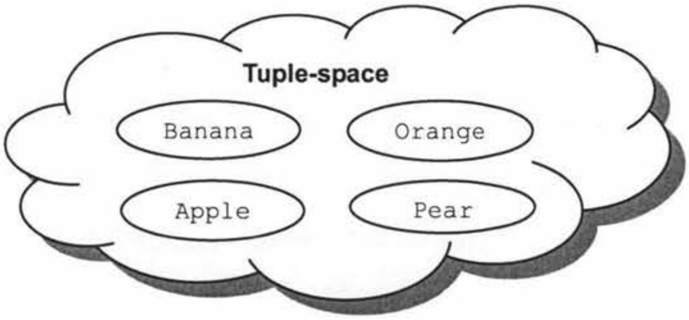


Figure4 – A Tuple-space consists of multiple subspaces

Users can define any type of object as the value of a tuple field as long as it implements `Serializable and tuplespace.interfaces.IField` interfaces and `equals()` method for matching algorithm. Thus, programmer has the flexibility to customize the matching process by defining how he wants that field value to be matched.

3.7 Retrieving Tuples from Tuple-spaces

When retrieving data from Tuple-space, the client presents a template to the Tuplespace server and gets matching tuple as return. The `rd(...)` and `tk(...)` operations take the same three parameters: A `Tuple` object as template; a `TransactionManager.TransactionProxy` object which represents the transaction under which the `rd(...)` or `tk(...)` operation is done; a `long` value representing how long the client is willing to be suspended and wait on response from the sever in milliseconds. If the timeout value is `TIMEOUT_FOREVER`, the client will be suspended forever until a matching tuple is found. If the timeout value is `TIMEOUT_NO`, the client will get the result immediately no matter if a matching tuple is found. Any retrieval operation can have only one matching tuple returned. If there are multiple tuples matching the template in the Tuple-space, the first found matching tuples is selected and returned to the client. The `tk(...)` operation is a consuming operation as it will remove the matching tuple from the space if it is found. In contrast, the `rd(...)` operation will only return a copy of the matching tuple when it is found.

Given the tuple instances shown in previous code samples, `t1` and `t2` have already been stored in the Tuple-space. The following code demonstrates how to retrieve a tuple that matches `template1` from the space:

```
try {
    Tuple result = ts.rd(template1, null, 5000);
    Tuple result2 = ts.tk(template1, null, 5000);
} catch (TransactionException e) {
    System.exit(0);
}catch (TupleSpaceException e) {
    System.exit(0);}
```

Code Sample4 – Retrieving tuples from the space

`result` and `result2` are exactly the same. Both of these two operations will wait for 5000 milliseconds if the tuple is not available immediately. However, after `result2` is returned, that tuple is removed permanently from the space.

3.8 Retrieving Fields from Tuples

A tuple returned from the Tuplespace server will normally be inspected for values and types of its fields. The fields of a tuple can be extracted into variables of type `tuplespace.Field`, A tuple offers the method `numberOfFields()` which can be combined with the method `get(int index)` in order to iterate over all fields in the tuple or to extract one particular field (fields are ordered). A `Field` object offers the methods `getValue()` and `getType()` which can be used for extracting field value and type.

To extract the integer value from the second field in `t1` (numbering starts at 0, so the index addressing the second field is 1), the following piece of code will do:

```
Class type = ((Field)result.get(1)).getType();
Integer value = (Integer)((Field)result.get(1)).getValue();
```

Code Sample5 – Retrieving field value and type from a tuple

The casting operator may cause a `ClassCastException` if the field is not of `Integer` type.

These are the main methods provided by the Tuplespace system to the client programmers. But with these simple methods, a great deal can be accomplished. Web servers, interactive chat systems, compute servers and online auction systems can all be created based on the Tuplespace system. Client/server tasks, message-passing, and other traditional programming models can also be emulated by the Tuplespace system.

4. Tuplespace Distribution Pattern

Considering the simple semantics for storing and retrieving tuples from a Tuplespace server, a wide range of distributed computing techniques are possible: client-server, unicast/multicast messaging, remote method invocation, etc. The proposed Tuplespace system uses a combination of client-server and message passing techniques to achieve the desired distributed computing. This chapter will present the details of the protocols deigned to form messages passed among the nodes in the system.

4.1 Message Passing Communications

Before we start discussing the distributed system, it is necessary to define what we meant by a *message*. Essentially, a message is a structured piece of information sent from one agent to another over a communication channel. Some messages are requests made to one agent by another (typically from clients to Tuplespace servers), some messages deliver data (typically, clients write tuples to the server, or the server returns tuples retrieved), other messages deliver notification to another agent (interaction among Tuplespace servers). In the proposed Tuplespace system, a typical message consists of a message identifier and a set of message arguments. The message identifier tells the receiver the purpose or type of the message. The arguments of the message contain additional data that will be interpreted accordingly based on the purpose of the message. They may contain the serialized form of a tuple object of an operation (e.g., the message "`TP_WRITE`" means: deserialize the following bytes as a tuple and store it to the associated subspace of the Tuple-space), or they may contain information used to carry out a request (e.g., "`TP_ABORT 1278812811`"means: Discard the locks held by transaction whose Id is 1278812811). By interpreting the messages properly, methods in the remote servers are also invoked to carry out the desired operations.

4.2 Communication Protocols

The messages passed in the proposed Tuplespace system are formatted by specially designed protocols. The protocols are designed to achieve the following goals:

- The data contained in a message has to be sufficient and efficient. The size of the message cannot be too big as the messages are sent across network, the bandwidth issues need to be taken into account.
- The message has to be well-defined to trigger method calls on remote Tuplespace servers to carry out desired operations.

As the research is focusing on Java implementation of a Tuplespace system, the low-level network protocols, such as TCP/IP protocols, will not be discussed, and the system works based on the assumption that the contents of the messages will be transferred accurately.

As explained in the previous chapter, the client needs a `proxy` to link to and access the services provided by the remote Tuplespace server, the process is shown in Figure5.

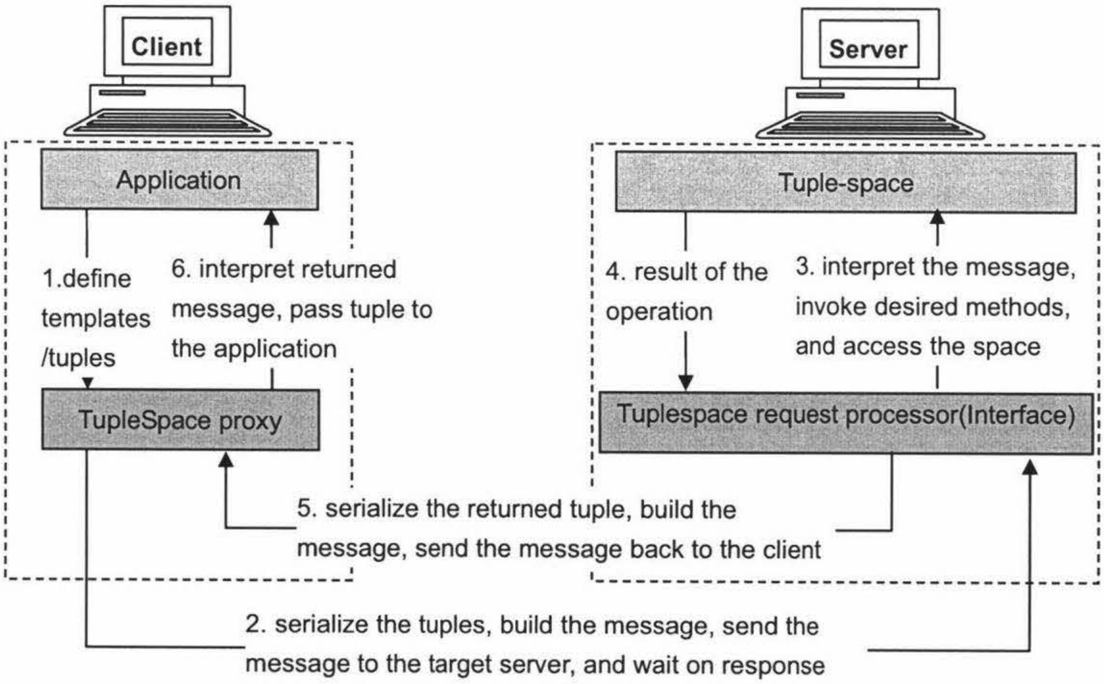
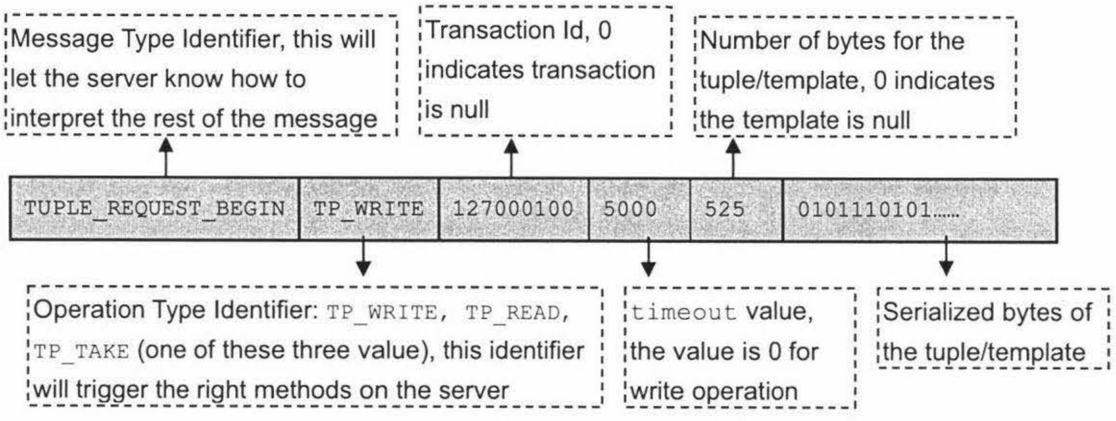

Client
Server
Application
Tuple-space
1.define templates /tuples
6. interpret returned message, pass tuple to the application
4. result of the operation
3. interpret the message, invoke desired methods, and access the space
TupleSpace proxy
Tuplespace request processor(Interface)
5. serialize the returned tuple, build the message, send the message back to the client
2. serialize the tuples, build the message, send the message to the target server, and wait on response

Figure5 – Message passing between Tuplespace client and server

As it's shown in Figure5, the Tuplespace system can be conceptually divided into a number of tiers doing certain tasks. Therefore, as long as each tier follows certain rules when it is interfacing the other tiers, any change made within the tier will not affect the others. And also, because the implementation detail of each tier is "isolated" from the others, the application programmer can focus on the development of the "real" application. The rules forming the communication between tiers are protocols.

An example of the protocol for the following three primary Tuplespace operations is shown in Figure6:

```
out(ITuple tuple, TransactionProxy transaction)
rd(ITuple template, TransactionProxy transaction, long timeout)
tk(ITuple tempalte, TransactionProxy transaction, long timeout).
```



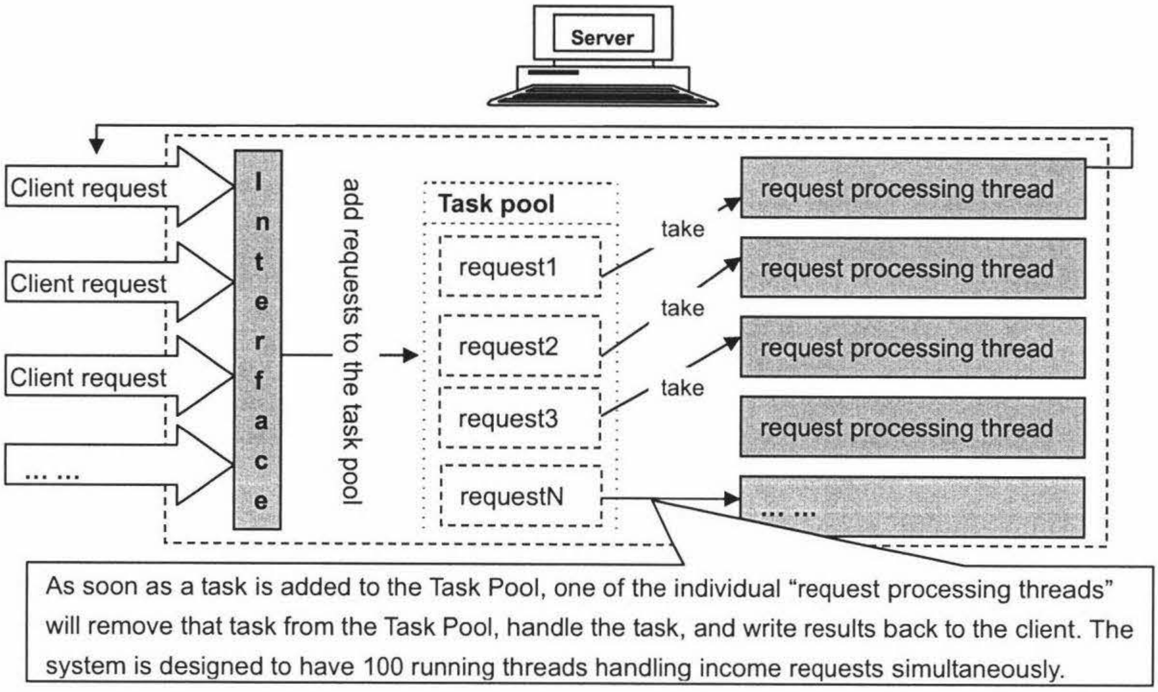
Message Type Identifier, this will let the server know how to interpret the rest of the message
Transaction Id, 0 indicates transaction is null
Number of bytes for the tuple/template, 0 indicates the template is null
TUPLE_REQUEST_BEGIN
TP_WRITE
127000100
5000
525
0101110101......
Operation Type Identifier: TP_WRITE, TP_READ, TP_TAKE (one of these three value), this identifier will trigger the right methods on the server
timeout value, the value is 0 for write operation
Serialized bytes of the tuple/template

Figure6 – Message protocol for `out( )`, `rd( )` and `tk( )` operations

Message Type Identifiers are simple, unique tokens that differentiate one type of message from another. They are just some simple byte values, where the agents on either end use a look-up table of some sort (In this case, the constant values of the Message Identifiers are defined in an interface named `TupleSpaceConstants.class`) to match the value with its meaning. Message arguments, on the other hand, are of many types. The message protocols used for this system are designed by using some basic data types, including integers, strings, short values, long values, and array of bytes, for message arguments. These arguments can be read and written directly using the `DataInputStream` and `DataOutputStream` classes with Java object serialization support. After the messages are sent to the server, now it's the server's job to interpret the message and behave accordingly. The message protocols used for the other actions, such as transaction creation, transaction completion and server cooperation, are of similar structure but with different arguments.

4.3 Asynchronous Message Handling

The Tuplespace system implements sever/client distribution computation model, which means that a server may often face the situation that multiple requests from different clients need to be handled simultaneously. The server is deigned to have 100 (This figure can be adjusted in the code) request-handling threads running in parallel (see Figure7). On the server side, there is an interface program that is listening to the communication port for incoming tasks. This interface program's job is simple and has to be done efficiently; otherwise it is going to be a potential bottleneck holding up the overall performance of the Tuplespace server. The interface program's duty is designed to focus on adding any incoming request to a "Task Poll"; it doesn't process the request itself at all. The request-handling threads that are running in parallel are all monitoring the "Task Pool" for new tasks. As soon as a new task is added to the pool, one of the idle threads will take that task out from the "Task Pool", interpret its content, invoke proper application objects, process the requests, and write response back to the client. The Tuplespace server uses Java's built-in synchronization mechanism to handle the concurrent access to the same system resource, so it is guaranteed that the same resource is accessed by one thread at the time. For example, when thread A is trying to retrieve from subspace "Apple", all the other threads that want to access subspace "Apple" have to wait until thread A finishes and releases the subspace.


Server
Client request
Client request
Client request
... ...
Interface
add requests to the task pool
Task pool
request1
request2
request3
requestN
take
take
take
request processing thread
request processing thread
request processing thread
request processing thread
... ...

As soon as a task is added to the Task Pool, one of the individual "request processing threads" will remove that task from the Task Pool, handle the task, and write results back to the client. The system is designed to have 100 running threads handling income requests simultaneously.

Figure7 – Multiple requests handling scheme

An example of the protocol controlling the format of the result returned from the Tuplespace server is shown as below:

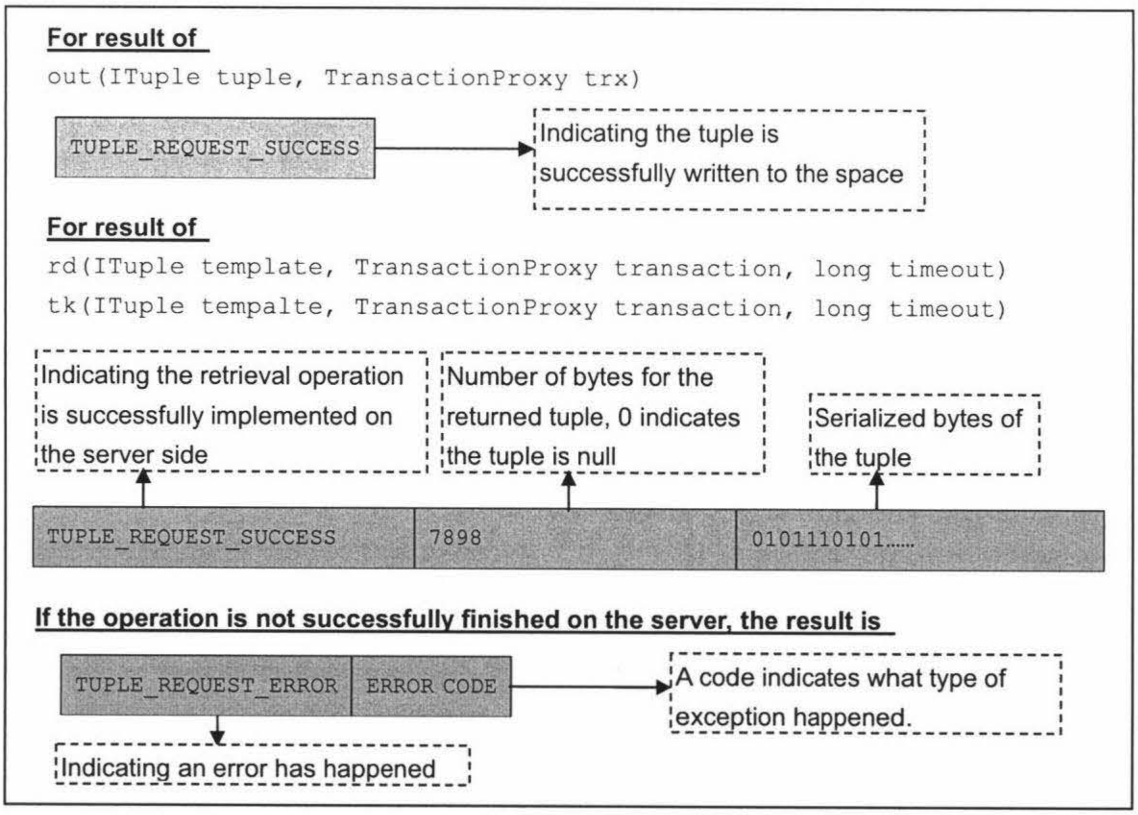

For result of
out(ITuple tuple, TransactionProxy trx)
TUPLE_REQUEST_SUCCESS
Indicating the tuple is successfully written to the space
For result of
rd(ITuple template, TransactionProxy transaction, long timeout)
tk(ITuple tempalte, TransactionProxy transaction, long timeout)
Indicating the retrieval operation is successfully implemented on the server side
Number of bytes for the returned tuple, 0 indicates the tuple is null
Serialized bytes of the tuple
TUPLE_REQUEST_SUCCESS
7898
0101110101......
If the operation is not successfully finished on the server, the result is
TUPLE_REQUEST_ERROR
ERROR CODE
A code indicates what type of exception happened.
Indicating an error has happened

Figure8 – Message protocol for formatting result returned from Tuplespace server

When an exception happens in remote server, the client needs to be informed about what kind of exception has happened. Ideally the exception happened in remote server can be risen at the client side so that appropriate exception handling actions can be called. Thus, when an exception happens at the server side, the server will look up for an associate code representing the exception and send the code back to the client within the response message. When the client receives the message containing the error code, it will call a method to throw the associated exception. Consequently, client's exception handing code is called to deal with the exception.

4.4 Synchronizing Operations

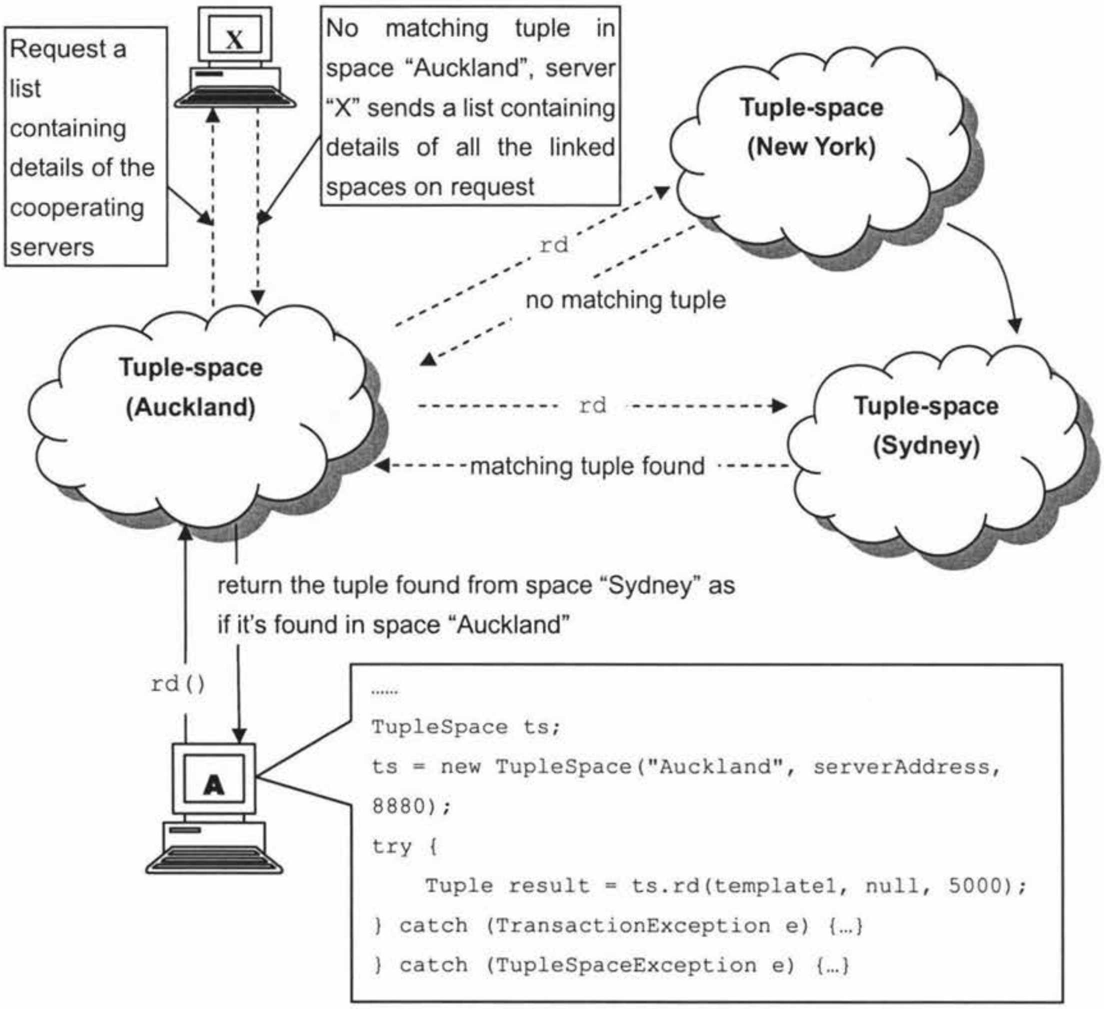
In a distributed environment, resources are frequently accessed by multiple processes simultaneously, so concurrency control is an essential issue need to be considered in distributed programming. The presented Tuplespace system has to take the precaution to ensure that operations on the same subspace are "thread-safe". That is, the system must ensure that one thread cannot modify the subspace while another thread is writing to or retrieving from the subspace. For instance, the system must ensure that `tk(...)` operation for one client is completed before the `rd(...)` operation for another begins, otherwise the `rd(...)` operation may read a tuple which is being removed or already removed. The system ensures thread-safety by enclosing accesses on the subspace within Java's `synchronized` code segments [25], [26], [27] as shown in the following chart:

```
synchronized (this) {
    result = lookupAccessibleTuple(atemplate, false, operation);
    while (result == null) {
        long deltaTime = targetTime - System.currentTimeMillis();
        if (deltaTime <= 0) {
            result = lookupAccessibleTuple(atemplate, false, operation);
            break;
        }else{
            try {
                ......
            } catch(InterruptedException ex) {
                System.out.println(ex.toString() + " occurred!"); }}}}
```

Java runtime ensures that only one thread can access the `synchronized` object at a time. So when one thread enters the `synchronized` block, accesses to that object (In our case, the object is the subspace) by other threads are blocked until the original thread exits the `synchronized` block or releases the subspace explicitly.

4.5 Collaborating Servers

The Tuplespace servers are linked and cooperating, a client's retrieval request will firstly be processed on the server that the client connects to explicitly(i.e. the primary Tuplespace server).


Request a list containing details of the cooperating servers
X
No matching tuple in space "Auckland", server "X" sends a list containing details of all the linked spaces on request
Tuple-space (New York)
rd
no matching tuple
Tuple-space (Auckland)
rd
Tuple-space (Sydney)
matching tuple found
return the tuple found from space "Sydney" as if it's found in space "Auckland"
rd()
A

```
......
TupleSpace ts;
ts = new TupleSpace("Auckland", serverAddress,
8880);
try {
    Tuple result = ts.rd(template1, null, 5000);
} catch (TransactionException e) {…}
} catch (TupleSpaceException e) {…}
```

Figure9 – Cooperating Tuplespace servers

For example, in Figure9, client A has a `rd(template1, null, 5000)` operation on

Tuplespace "Auckland", so Tuplespace "Auckland" is the primary Tuplespace server and it checks if a matching tuple can be found within its own space. If a matching tuple is found, then a response will be sent back to the client and the request is fulfilled successfully. If no matching tuple could be found from the primary Tuple-space, then Tuplespace server "Auckland" will send a request to a special server which holds a list of all the cooperating Tuplespace servers. When Tuplespace server "Auckland" receives the list, it will then send `template1` to the other Tuplespace servers (i.e. alternative Tuplespace servers) in the list one by one until a matching tuple is found or time runs out. As shown in Figure9, "Auckland" first sends the template to "New York", "New York" responds with "No matching tuple found". After that, "Auckland" sends the template to the second server in the list which is "Sydney", "Sydney" responds with a matching tuple. Finally, "Auckland" returns to client A a matching tuple found. The entire cooperation among the servers is invisible to the clients. The advantage of having a server managing the list of the cooperating Tuplespace servers is that it is easy and efficient to manage. As the list is maintained at a center location, the system administer can easily add new cooperating servers to the system or delete one from the system, or update its details. The distributed Tuple-spaces are shared, so the client is able to access the tuples in all the spaces. Plus, by having cooperating servers, this can help with latency issues as client programmer can place tuple to the server which is physically closer to the client computer or can provide faster processing for complex tuples. The cooperating server design also helps with improving the persistency and stability of the system. The contents of the Tuple-spaces can be transferred and copied to the other servers, so if a Tuplespace is shutdown for maintenance purpose or it crashes for no reason, its data can still be available to the users.

4.6 Design Patterns

As it was briefly mentioned in Chapter 2, the "space" architecture is able to offer simpler design, and more robust results that are easier to maintain and integrate comparing to other models for developing distributed applications. This section is going to present some details and samples about how applications can benefit from using the "space" approach.

The master-worker pattern nicely exemplifies the use of the Tuplespace model in developing a web application. Client requests are received by a web server, which sends them asynchronously, as request objects, to the Tuple-space. The responsible worker processes then react in parallel, processing the requests by retrieving the request objects from the space and returning the results into the space as answer objects. The web server, upon notification of answers, then serves them to the clients, repackaging as required by the user interface, e.g. in HTML or XML format.

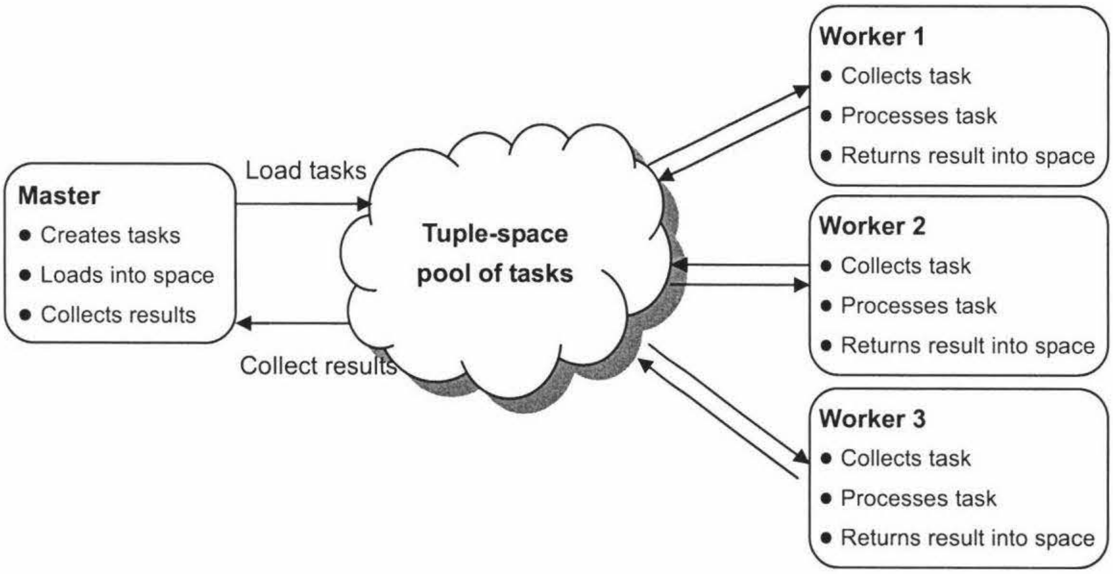


Figure10 – Master-worker programming pattern

By using a space as communication medium and the decoupling of the interacting processes, this implementation scales easily. In the event of a big increase in the number of simultaneous requests, additional web server and worker machines (cheap standard hardware) can simply be attached to the system as needed

4.7 Data Package Switching

The presented system is built based on the assumption that the content of each message is accurately transferred between the nodes in the network. The system uses TCP/IP as the underlying network protocol and requires that any pair of nodes (server-server, client-server) wishing to interact is able to send messages to each other in both directions. This system also assumes that the messages are of arbitrary length. The system permits the underlying network protocol to split a message up into smaller pieces for transmission, and to reassemble the message before presenting it to the Tuplespace server. If necessary, explicit packetization could be added later on, such as validating the checksum. A good message is one with a good checksum. Messages with bad checksums are simply discarded or maybe re-sent, so bad messages are eliminated. Packetization is omitted in the presented system to make the algorithms simpler.

With the well-formed messages, application programmers can focus on the classes making up the application around application issues, not issues related to the communication scheme they happen to be using. Likewise, the communications subsystem can be designed and updated independently, based on the communication needs of the overall system.

5. Transactions

The proposed Tuplespace system includes a sub-package `tuplespace. server.transaction` to provide basic atomic transactions that group multiple operations into a single atomic operation. This chapter introduces what transaction is and the ACID properties that are provided by transactions. How the Tuplespace system achieves transaction management and how the transactional operations are implemented are explained in this chapter as well.

5.1 Transactions and ACID Properties

Transactions are a fundamental tool for many kinds of computing[28]. A transaction allows a set of operations to be grouped in such a way that they either all succeed or all fail; the operations in the set appear from outside the transaction to occur simultaneously.

Transactional behaviors are especially important in distributed computing systems such as this Tuplespace system, where they provide a mean for enforcing consistency over a set of operations that are performed on remote computers. For the operations that are members of a transaction, one response to a remote failure is to abort the whole transaction, thereby ensuring that no partial results can be written to the space.

The ACID properties [29] traditionally offered by database transactions are preserved in transactions on the Tuplespace system. The ACID properties are:

- Atomicity: All the operations grouped under a transaction occur or none of them do.
- Consistency: When a transaction completes, it must leave the system in a consistent state. Consistency includes issues that only matter to humans. Such as in an airline scheduling example, a Flight object that should always have a Pilot object associated with it. The enforcement of consistency is outside of the

control of the transaction itself – a transaction is a mean to maintain consistency but it doesn't guarantee consistency itself.

- Isolation: Ongoing transactions should not affect each other. The client in a transaction should see only intermediate states resulting from the operations within its own transaction, not the intermediate states of any other transactions. For example, if a tuple is written inside a transaction, it is only visible to the operations within the transaction. None of the operations from outside of the transaction knows the existence of the tuple.
- Durability: The results of a transaction should be as persistent as the entity on which the transaction commits. However, such guarantees are up to the implementation of the entity, in this case, the lifetime of the Tuplespace service determines the persistence of the transaction results.

Transactions implemented in this Tuplespace system differ from single-system (a system that only runs within the scope of a single computer) transactions or distributed transactions. The clearest difference with a single-system transaction is that single-system transactions are maintained locally within one node. In the proposed Tuplespace system, the transaction objects are actually remote objects that reside on a remote Tuplespace server. The remote transaction model implemented in this Tuplespace system is discussed in next section.

The remote transaction implementation is also different from the distributed transaction model implemented in Sun's JavaSpaces[30], [31]. In the remote transaction model, it appears to any client that all operations performed under a transaction have occurred or none have, thereby achieving isolation. In other words, no client will ever see only part of the changes made under a transaction. In a distributed transaction model, a transaction can span multiple servers/processes, so it is possible for a client whose transaction spans three Tuplespace servers to see the committed state of the transaction in one server and the pre-committed state of the same transaction in the other servers. In other words, one transaction may contain different operations and each takes different amount of time to be committed on different sever. Distributed transactions often implement two-phase locking (such as Sun `net.jini.core.transaction`) and transaction manager to coordinate the participants of a transaction [32].

5.2 Distributed Transaction Model

As the two-phase commit protocol is widely adapted as the distributed transaction model in distributed systems[16], [29], this section is going to provide a brief overview of this protocol. The two-phase commit protocol defines the communication patterns that allow ***distributed*** objects to group a set of operations in such a way that they appear to be a single operation. This protocol requires a kind of transaction manager object that can enable all participants of a transaction to know whether they should commit the operations or abort them. A participant can be any object that supports the participant contract by implementing the appropriate interface. Under the two-phase commit protocol, a transaction completes when any entity either commits or aborts the transaction. Commit a transaction successfully means all operations performed under that transaction are completed. Aborting a transaction means that any operations performed under that transaction will appear never to have happened.

In Sun's JavaSpaces, the two-phase commit protocol requires each participant to vote when committing a transaction, where a vote is either prepared (ready to commit), not changed (read-only), or aborted (the transaction should be aborted). If all participants vote "prepared" or "not changed," the transaction manager will tell each "prepared" participant to roll forward, thus committing the changes. Participants that voted "not changed" need do nothing more. If the transaction is aborted, the participants have to roll back any changes made under the transaction.

5.3 Remote Transaction Implementation

In this distributed Tuplespace system, clients' requests are all encoded into messages and sent over a communication network to be implemented on a remote server, so are the transactional operations. When the client wants to create a transaction, this request is sent to the Tuplespace server. On the server side, the transaction manager creates a `Transaction` object with a unique Id and an associated `TransactionProxy` object, and the server returns the `TransactionProxy` object back to the client. Each Tuplespace server has a transaction manager maintaining all the transactions running on the server. The transaction proxy object is a "representative" of the actual transaction object on the client side and it contains the same Id as the transaction object it represents. Another key difference from distributed transaction model is that though the transaction objects are associated between the server and the client, they are not shared between processes. In other words, a process cannot join a transaction which was created by another process.

After a client has a transaction object in hand, it can start transactional operations. The code is shown in the following table:

```
//continuing from previous code
TransactionManager.TransactionProxy trx = null;
try {
    trx = ts.createTransaction();
}catch (TransactionException e) {
    System.out.println(e);
}

ITuple result = null;
try {
    result = ts.rd(template, trx, 5000);
    result = ts.tk(template, trx, 5000);
} catch (TransactionException e) {
    System.exit(0);
}catch (TupleSpaceException e) {
    System.exit(0);
}
```

Code Sample6 – Create transaction object, and perform transactional operations

`InvalidTransactionStateException` will be thrown if the client wants to do an operation when the transaction is not at the right state. For example, when the transaction is in the middle of committing process or already committed, if the client wants to do more operation under the transaction, this exception will be thrown. `InvalidTransactionStateException` extends from `TransactionException`.

`TransactionNotExistException` will be thrown if the server-side transaction object represented by the transaction proxy object no longer exists on the server. `TransactionNotExistException` extends from `TransactionException`.

For any other exceptions related to transaction, `TransactionException` will be thrown.

5.4 Transactional Operations

Single-level transactions affect operations in the following ways:

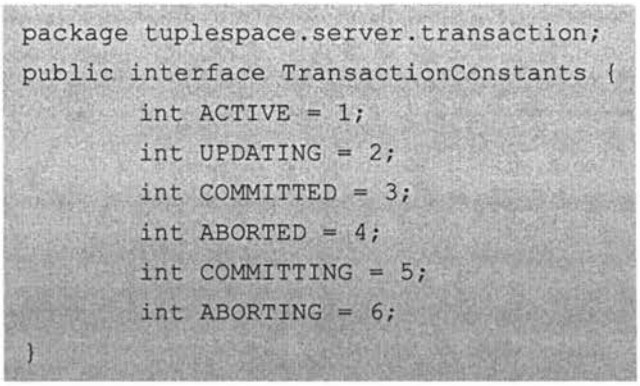
- `out(...)`: A tuple that is written is not visible outside its transaction until the transaction successfully commits. If the tuple is taken within the transaction, the tuple will never be visible outside the transaction and will not be added to the space when the transaction commits. Tuples written inside a transaction that aborts are discarded.
- `rd(...)`: A read operation matches any tuple written under that transaction and in the entire primary Tuple-space and in the alternative Tuple-spaces. `rd(...)` operation within a transaction is designed to look for matching tuples written inside the transaction first, then the primary Tuple-space, then the alternative spaces. The reference to the matching tuple found by the provided transaction is added to a list maintained within the provided transaction. By recording the references to the locked tuples in a list, the transaction is able to quickly locate the tuples that are locked by it and manage the locks. Such a tuple can be read by any other transaction to which the tuple is visible, but cannot be taken by another transaction. The matching tuple found will be marked with a lock object of `tuplespace.TupleLock` class. A lock object contains the locking transaction's Id

and the locking mode, in this case, the locking mode is set to "READ_MODE" as the tuple is locked by a read operation of a transaction. When a tuple is locked by a transaction under "READ_MODE", the locked tuple can only be taken by the locking transaction. If the matching tuple is found from an alternative server, the matching tuple will be locked the same way as it is found in the primary server. A copy of the matching tuple found from the alternative server will be returned to the initial server (i.e. the primary server) and added to a special list kept inside the transaction object in the initial server. The special list is maintained for two purposes: it keeps record of the alternative Tuplespace servers that the transaction has retrieved matching tuples from. When the transaction commits/aborts, the primary Tuplespace server needs to notify every alternative Tuplespace server recorded in this list to release the tuples locked by the transaction; the list acts as a cache containing the tuples read from alternative Tuplespace servers. For future rd(...) operations under the same transaction, the Tuplespace server will search the cache for matching tuple before troubling the alternative Tuplespace servers, so this eliminates the need to send message over the network and ask each of the alternative servers to search again. If a transaction commits/aborts, the read locks held by it will be discarded and the locked tuples are set back to its original states.

- tk(...): A take operation matches like a read with the same template. When taken, the reference to the tuple is added to the list recording the tuples taken by the provided transaction and the list is maintained within the transaction object. Such a tuple can not be read or taken by any other transaction. The matching tuple is locked by a TupleLock object which contains the locking transaction's Id and the locking mode, in this case, the tuple is locked under "TAKE_MODE". If a matching tuple is found already locked under "READ_MODE" by the provided transaction, the lock is updated to "TAKE_MODE". The reference to the tuple will be shifted from the list recording read tuples to the list recording taken tuples. Tuples locked under "TAKE_MODE" are not visible to any operations including the operations under the same transaction as if they are already completely removed from the system. If the matching tuple is found from an alternative Tuple-space, the matching tuple will be locked the same way as it is found in the primary server. A copy of the matching tuple found from alternative server will be returned to the primary server and added to its special list which is also used for `rd(...)`. By locking the taken tuple instead of removing it from the space, if the server crashes because of some reasons, the taken tuples are still kept in the space (the "permanent" memory) and server has the chance to recover them and roll back the operations. If the transaction commits successfully, the tuples taken by the transaction from the space are removed permanently. If the transaction aborts, the locks are discarded as the `tk(...)` operations have never happened. If tuples written under a transaction are taken, the tuples won't be written to the space.

5.5 Transaction States

Every transaction object is marked with a state value at any time. The state value indicates the current status of the transaction and determines what operations can be performed under it. `TransactionConstants.class` file defines state constants, see Code Sample7.



```
package tuplespace.server.transaction;
public interface TransactionConstants {
        int ACTIVE = 1;
        int UPDATING = 2;
        int COMMITTED = 3;
        int ABORTED = 4;
        int COMMITTING = 5;
        int ABORTING = 6;
}
```

Code Sample7 – Transaction state constants defined in the system

When a transaction is created, its initial state is `ACTIVE`. `ACTIVE` is the only valid state for the transaction manager to invoke the transaction object for any operation. For example, when the transaction's state is `COMMITTED` or `ABORTED`, this means that the transaction is already completed and no more operation can be done under this transaction. When the transaction's state is `COMMITTING` or `ABORTING`, this means that the transaction is in the completion process, again, no more operation can be done under this transaction.

5.6 Completing a Transaction

A transaction can be completed with code shown as below:

```
......
//complete a transaction
try {
    ts.commitTransaction(trx);
    ts.abortTransaction(trx1);
}
catch(TransactionNotExistException e)
{
    System.out.println(e);
}
catch(TransactionCannotCommitException e)
{
    System.out.println(e);
}
catch(TransactionCannotAbortException e)
{
    System.out.println(e);
}catch(TransactionException e)
{
    System.out.println(e);
}
```

Code Sample8 – Transaction completion syntax

The `commitTransaction()` and `abortTransaction()` operations are both one parameter methods, the only parameter needed is a `TransactionProxy` object which represents the actual transaction object located on the remote Tuplespace server. Both methods have to be called within a `try` clause since exceptions may happen.

Before we start discussing about the details of the completion process, let's have a look at the internal structure of a typical transaction object as shown in Figure11:

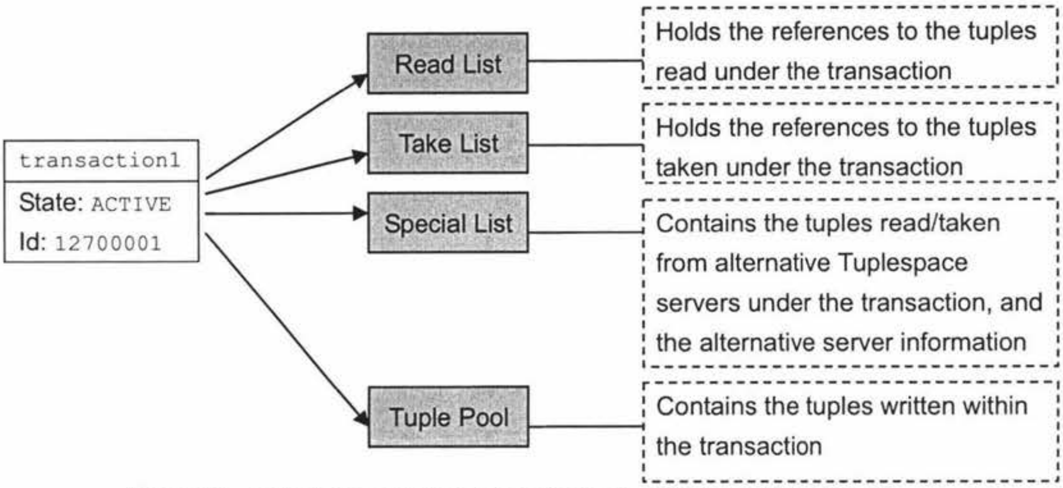

transaction1
State: ACTIVE
Id: 12700001
Read List
Take List
Special List
Tuple Pool
Holds the references to the tuples read under the transaction
Holds the references to the tuples taken under the transaction
Contains the tuples read/taken from alternative Tuplespace servers under the transaction, and the alternative server information
Contains the tuples written within the transaction

Figure11 – The internal structure of a typical transaction object

If the transaction is asked to be committed, the following steps need to be performed:

1. Check the state of the transaction. If the state is `UPDATING`, then wait until the update process is completed. If the state is `COMMITTING`, or `COMMITTED`, or `ABORTING`, or `ABORTED`, this means that the transaction is already completed or under the completion process, so no more operation can be performed and an exception will be thrown.
2. When the transaction state is `ACTIVE`, the commit operation will lock the transaction by marking its state `COMMITTING`, so no further operations can be added to the transaction.
3. Go through the list recording the references to the tuples read from the primary Tuple-space under the transaction, release the locks so they are set back to normal status and can be taken by other operations. Clear this list.
4. Go through the list recording the references to the tuples taken from primary Tuple-space under the transaction; permanently remove the tuples from the space. Clear the list.
5. Go through the list containing the tuples read or taken from alternative spaces under the transaction, send a notification message to each of the alternative spaces that contain the read/taken tuple to inform them to release the read locks

and permanently remove the tuples taken by the transaction from their spaces. Clear the list.

6. Go through the list containing the tuples written under the transaction; write each tuple to the local space. Clear the list.
7. Set the transaction state to COMMITTED.

Note: the commit process is done in a durable fashion. When the transaction manager is going through the steps to commit the transaction, any change it makes to the space is recorded in a durable container. Therefore, if an exception happens at a certain point during the committing process, the transaction manager can rollback all the changes that have been made till that point. If any exception happens, the abort method will be called.

If the transaction is asked to be aborted, the following steps are performed:

1. Check the state of the transaction. If the state is UPDATING, then wait until the update process is completed. If the state is COMMITTING, or COMMITTED, or ABORTING, or ABORTED, this means that the transaction is already completed or under the completion process, so no more completion operation can be performed under the transaction and an exception will be thrown.
2. When the transaction state is ACTIVE, the abort operation will lock the transaction by marking its state ABORTING, so no further operation can be added to the transaction.
3. Go through the list containing the references to the tuples read from the primary Tuple-space under the transaction; discard the locks so that they are set back to normal status as if the operations under the transaction have never happened. Clear this list.
4. Go through the list containing the references to the tuples taken from the primary Tuple-space under the transaction; discard the "TAKE_MODE" locks held by the transaction so the locked tuples are set back to their normal status as if the operations under the transaction have never happened. Clear the list.

5. Go through the list containing the tuples read or taken from alternative spaces under the transaction; send a message to each of the alternative spaces that contain the read/taken tuples to notify them to discard the "READ_MODE" and "TAKE_MODE" locks held by the transaction on those alternative servers. Clear the list.
6. Clear the pool containing the tuples written under the transaction.
7. Set the transaction state to ABORTTED.

Note: the abort process is also done in a durable fashion. When the transaction manager is going through the steps to abort the transaction, any operations aborted are recorded in a durable container. Therefore, if an exception happens at a certain point during the aborting process, the transaction manager can try to abort the rest of the provided transaction starting from the failure point.

Durability is a commitment, but it is not a guarantee. It is impossible to guarantee that any given piece of stable storage can *never* be lost; one can only achieve decreasing probabilities of loss. Data that is written to a disk may be considered durable, but it is less durable than data saved to two or more separate, redundant storages. When referring to "permanent storage" in this system, it means the actual space containing the tuples, and the space lifetime is determined by the Tuplespace service. When a Tuplespace service is shut down, anything in the space will be lost unless the content of the space is copied to somewhere else beforehand.

Transaction support is very useful when building applications with high reliability and consistency requirements, however, just one level transaction may not be efficient at some situation. Next chapter will be discussing about Nested Transactions which offer a number of extra features than single-level transactions.

6. Nested Transactions

In the previous chapter, the notion of transaction was presented along with the techniques that have been used to achieve ACID properties in the proposed Tuplespace system. This chapter extends the single-level transaction idea by introducing Nested Transactions (also referred to as subtransactions). The purpose of this chapter is to present the details of how Nested Transactions are implemented in the system. Firstly, we will start with explaining what nested transactions are.

6.1 What are Nested Transactions?

Transactions are very useful and help solve many problems in both centralized and distributed computing situations. However, there are several situations that could be better solved by adding Nested Transactions.

A nested transaction is a new transaction that begins from within the scope of another transaction[28]. Nested transactions are an extension of transactions. The difference between transactions and nested transactions is that nested transactions have a more complicated internal structure. A transaction is just a group of operations that are performed as a unit. Nested transactions have a hierarchical grouping structure: each nested transaction consists of zero or more operations and possibly some nested transactions.

Nested transactions offer several extra features, including:

- Nested transactions allow programmers to sub-divide a complex operation into a number of smaller and simpler operations.
- Nested transactions enable errors to be isolated within a certain scope.
- Nested transactions can operate concurrently.

Nested transaction model enables composition of multiple transactions into a single new transaction and concurrency control is provided within the transaction. In addition to solving problems of concurrent access within transactions, nested transactions can provide a method for maintaining better robustness of the system by limiting the effects of a failure to a small part of a transaction[28]. For example, suppose we wish to perform a transaction, consisting of a number of nested transactions each doing something different. As the number of nested transactions increases, the probability of failure increases, such that the top-level transaction's probability of success goes closer and closer to zero. However, if we treat each nested transaction as a full-fledged transaction within the scope of its containing transaction, then failure of one of the nested transactions needs not affect the results of any others. If the client requires that all nested transactions perform the requested operation, then a failed nested transaction can be retried until it succeeds. The advantage is that only the failed nested transactions need to be redone. Thus each nested transaction (at any nesting level) acts like a firewall, protecting its internal operations against the affects from outside influences. Also, the failures inside each nested transaction are shielded within a certain scope and won't affect the outside world.

When a nested transaction completes successfully, it will be said to have committed, even though it is not a top-level transaction. Certainly, this kind of commitment is relative: any updates become permanent only if all the nested transactions containing the committed nested transaction also commit, and the enclosing top-level nested transaction completes successfully. Thus, top-level transactions are special: they are the only irrevocable transactions in nested transaction hierarchy. Abort operation on one nested transaction doesn't force any of the containing nested transactions must abort as well. Aborting is always irrevocable, that is, an aborted transaction's operations must be rolled back. The details of the relationship of committing and aborting of nested transactions to committing and aborting of their containing transactions, including the manipulation of locks and tuple state restoration, form the main content of this chapter.

6.2 Some Terminology

Before proceeding to the details of the implementation of nested transactions, it is useful to introduce some terminology.

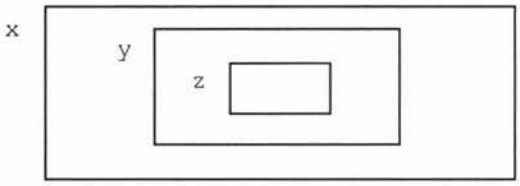


Figure12 – A transaction nesting diagram

Figure12 illustrates three transactions, x, y and z. The contours indicate that x has greater scope than y, that is, that y is a nested transaction of x. Likewise z is a nested transaction of y. The contours emphasize the concept that each nested transaction is a miniature universe of full functions. Contours will never intersect, because a nested transaction's effects and lifetime are always strictly bounded by its containing transactions(if any). The multi-level transaction relationship can also be described with tree diagram as shown in Figure13.

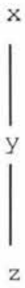
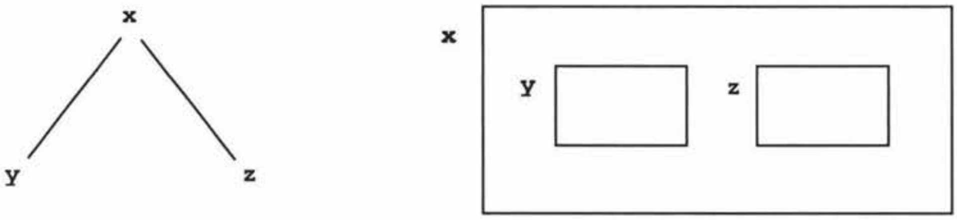


Figure13 – A tree diagram for transaction nesting

If x had two subtransactions, either of the diagrams in Figure14 would describe the situation:



Tree Diagram

Nesting Diagram

Figure14 – Transaction nesting

Since transaction relationships follow trees, this thesis will use tree terminology for familial relationships, to express transaction relations. Thus, transactions having subtransactions could be called parent transaction, and their subtransactions are their child transactions.

6.3 Synchronizing Nested Transactions

We will first introduce the locking principles used in the design, and make the extension to the nested transactional operations later.

In single-level transaction as discussed in the last chapter, if a tuple is locked by a transaction, then the locking transaction has exclusive access to the locked tuple until the transaction commits or aborts, and no other transaction can lock the tuple object for that period of time. However, additional mechanism and rules are needed to handle such case in nested transactions. Here is an example. Suppose there are some transactions related as in Figure15:

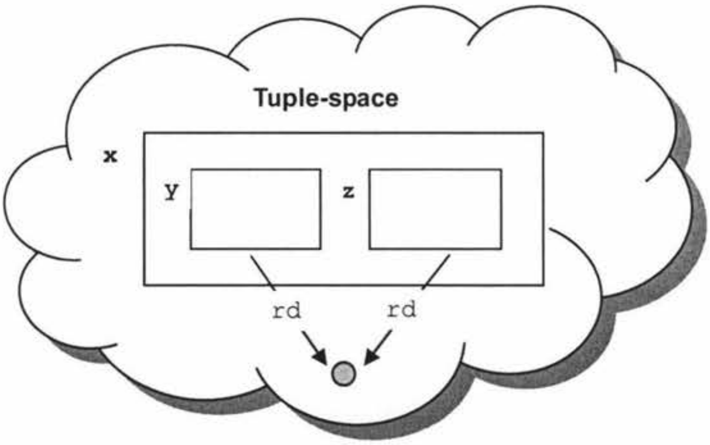


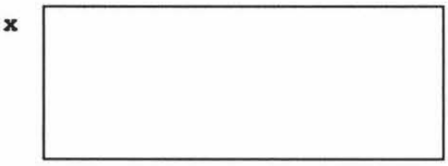
Figure15 – Transaction nesting example

Further suppose that y and z are processed serially and both try to read the same tuple object. (There is no conflict because these two transactions' lifetimes do not overlap.)

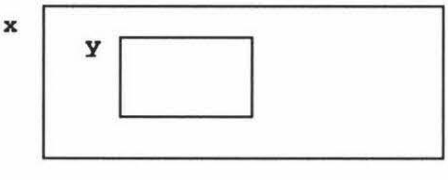
A first observation is that when y commits, the read lock on the tuple cannot be released entirely. The reason is that x can still abort, undoing y's operation. In order to insure consistency, the locking mechanism must make sure that transactions outside of x cannot take over the control on the tuple until x commits. The solution is to "pass" the lock from y to x when y commits, thus nothing outside of x can lock the tuple object. At this point, the situation may get complicated if z wants to acquire the lock, as it's said that x is currently holding the lock that inherited from y.

In nested transaction, the same type of lock is used as for single-level transactions. In the presented system, a tuple could be locked by a nested transaction under either "READ_MODE" or "TAKE_MODE". If a lock is set on a tuple, it means that there is a transaction "holding" the lock and the lock "holding" transaction has exclusive control of the tuple. By saying "holding" the lock, it actually means that the lock is marked with the transaction's Id. There can be at most one holder of a lock at a time. When a nested transaction commits, its direct parent transaction will retain all locks held by the committing child (by updating the lock Ids to the direct parent transaction's Id). When locks move from a committed child to the parent, it indicates that the parent has inherited the locks. In actual implementation, when a child transaction commits, all the locks with the child transaction's Id will be updated to its direct parent transaction's Id, and the references to the locked tuples stored in the child transaction's lists will be transferred to the direct parent transaction's lists as if the direct parent transaction is holding the locks now. Figure16 shows the sequence of situation as y, then z and lastly x run and commit; the relationship of three transactions were shown in Figure15.

1. Transaction x is created and running.



2. Transaction x creates a child transaction y.



3. Transaction y reads a tuple from the space, locks it and holds the lock.
4. Transaction y commits, so transaction x holds the lock.
5. Transaction x creates another inferior transaction z.
6. Transaction z reads the same tuple which is currently locked by x. Now, z holds the lock. This is permitted because z is the inferior of the only holder of the lock, x.
7. Transaction z commits, the lock is inherited again by x.

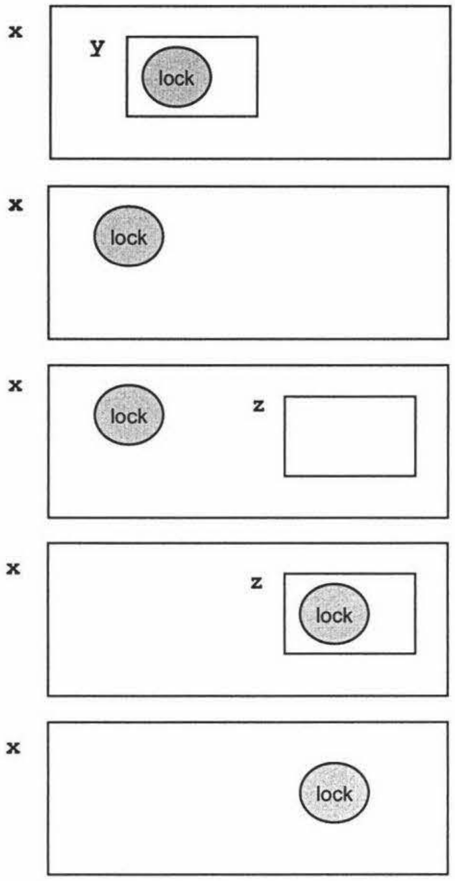


Figure16 – Lock moving among nested transactions

This little example demonstrates what to do if a child transaction commits. In the meantime, if another operation outside x wants to read the tuple which has been locked, that transaction can get a copy of the tuple, but the lock is still held by x or its child transactions. This also means that no any other operation outside x can remove the locked tuple during this period. When the nested transaction aborts on purpose, all the locks held by it will be released because no any effect on permanent storage is meant to happen. If the nested transaction aborts due to exceptions happened, because this Tuplespace system omits special handling of nested transaction exceptions, locks will also be released as normal aborts.

Here are the locking rules used for nested transactions in the system:

- A nested transaction can hold a lock of "READ_MODE" on a tuple if no other transaction holds the lock in "READ_MODE" or "TAKE_MODE", or the current holder of

"READ_MODE" lock on the tuple is a superior of the requesting nested transaction.

- A nested transaction may hold a lock in "TAKE_MODE" on a tuple object if no other transaction holds the lock in "TAKE_MODE", and no any other transaction except one of its parent transactions is holding the lock in "READ_MODE".
- When a nested transaction aborts, all its locks (of all modes) are discarded.
- When a nested transaction commits, all its locks (of all modes) are passed to its direct parent transaction (if there is any). This means the parent transaction holds each of the locks (in the same mode as the child transaction held).

When a nested transaction holds a "TAKE_MODE" lock on a tuple, it prevents any other transaction (including its child transactions) from accessing the locked tuple, and holding a "READ_MODE" lock prevents the tuple from being taken by any transactions except the lock holding transaction itself or its child transactions. Inheritance of "TAKE_MODE" locks when a nested transaction commits has two effects: it permits the child transactions within the parent's scope to see any changes, and have the chance to abort the changes; and it prevents transactions outside the scope from either reading or taking the tuple. Inheritance of "READ_MODE" locks when a transaction commits prevents removal by operations outside the parent transaction's scope. This insures that the parent transaction is always presented with tuples at consistent status.

The above rules assume that a transaction doesn't commit until all its child transactions were terminated (so that it retains any necessary locks). In other words, the lifetime of a nested transaction is always contained in its parent transaction's lifetime.

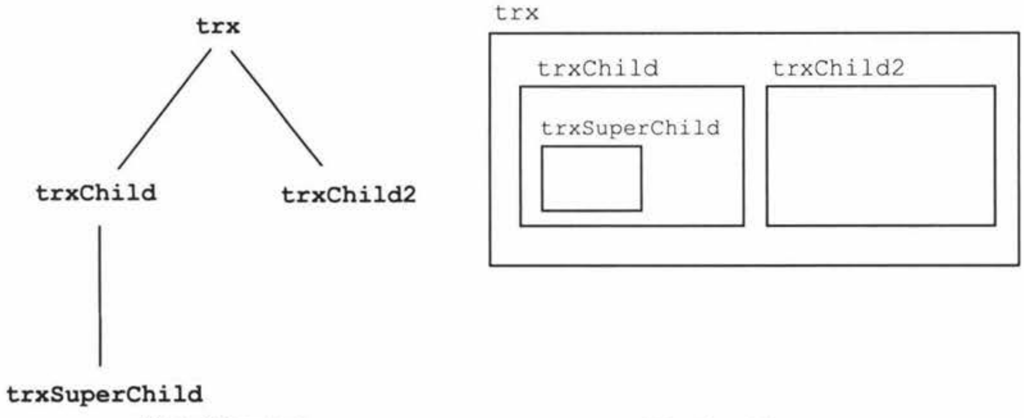
6.4 Create Nested Transactions

The following code demonstrates how to create multi-level nested transactions.

```
TransactionManager.NestableTransactionProxy
                                             trx = null,
                                             trxChild = null,
                                             trxChild2 = null,
                                             trxSuperChild = null;
try {
    trx = ts.createNestableTransaction(null);
    trxChild = ts.createNestableTransaction(trx);
    trxChild2 = ts.createNestableTransaction(trx);
    trxSuperChild = ts.createNestableTransaction(trxChild);
}catch (TransactionException e) {
    System.out.println(e); }
```

Code Sample9 – Creating nested transactions

From the client's point of view, any transactions contained inside a nested transaction hierarchy needs to be of type `NestableTransaction` and created by using `createNestableTransaction(TransactionManager.NestableTransactionProxy parentTrx)`, including top-level transactions. As shown in Code Sample9, transaction `trx` is a top-level transaction, and all the other transactions are its children and the parent transaction's proxy object has to be passed in as a parameter to define the relationship. The relationships among these transactions are illustrated in Figure17.



Tree Diagram **Nesting Diagram**

Figure17 – Relationship diagram of nested transactions

Like single-level transactions described in the previous chapter, client is working with a `NestableTransactionProxy` object which is transparently interacting with the actual `NestableTransaction` object on the server.

Like single-level transactions, nested transactions have the following properties:

- A transaction is serializable with respect to its siblings (nested transactions at the same level and contained by the same parent transaction). Accesses to shared resources by sibling transactions have to obey the synchronization rules.
- A nested transaction is a unit of recovery by itself. A nested transaction can be committed/aborted independently of its siblings.
- A nested transaction is a unit of atomicity by itself. Either all or none of the effects of a nested transaction's operations can take place.

In addition, they have the following features which make them different from single-level transactions:

- Nested transactions can have parents and children. A nested transaction's operations are not considered to conflict with its parent transactions'. Thus, it can take over a tuple which has been locked by its parent under "`READ_MODE`".
- When a child transaction aborts, it does not automatically abort the parent transaction. The parent transaction is free to perform other operations.
- When a child transaction commits, it releases the locks held by it to its direct parent transaction and makes its actions be part of the action set of its direct parent transaction. Thus, when the direct parent commits, it commits not only the operations it performed directly but also those performed by its child transactions.
- A parent transaction's operations are considered to conflict with its child transactions'. Thus, it cannot take over a tuple if a child transaction's lock prohibits this. For example, if a tuple is locked by a child transaction under "`READ_MODE`", the parent transaction can only read it without taking over the control of the lock.
- A child nested transaction is not a unit of consistency or durability since it does not on its own leave the Tuplespace in a consistent state. The commit result of a child transaction is relatively persistent, because it depends on the top-level transaction to commit successfully.

If one of a nested transaction's ancestors aborts, it doesn't matter whether the transaction aborts or commits; even if the transaction commits, its effects will be undone by the abortion of its ancestor. Thus, the *durability* property of the traditional ACID properties needs to be modified for a nested transaction situation: the effects of a committed top-level transaction and those of its committed descendents are not undone by a failure[28].The presented system requires that all of a transaction's children must be resolved before the transaction can attempt to commit. A nested transaction may abort at any time, and all of its child transactions' actions will be aborted as well.

6.5 Nested Transaction Operations

Nested transactions affect operations in the following ways:

- `out(...)`: A tuple that is written is not visible outside its transaction until the transaction successfully commits. If the tuple is taken within the transaction, the tuple will never be visible outside the transaction and will not be added to its parent transaction(if there is any) or the space when the transaction commits. Tuples written under a nested transaction that commits are added to the Tuple-space if the committing transaction is a top-level transaction; otherwise the tuples are written to its direct parent transaction. Tuples written under a nested transaction that aborts are discarded.

- `rd(...)`: A read may match any tuples written under that nested transaction, any of its parent transactions, the primary Tuple-space and the alternative Tuple-spaces. The Tuplespace server is designed to match tuples written inside the transaction first, then its parent transactions(if any), then the primary Tuple-space, and finally the alternative Tuple-spaces. The searching order is shown in the following diagram.

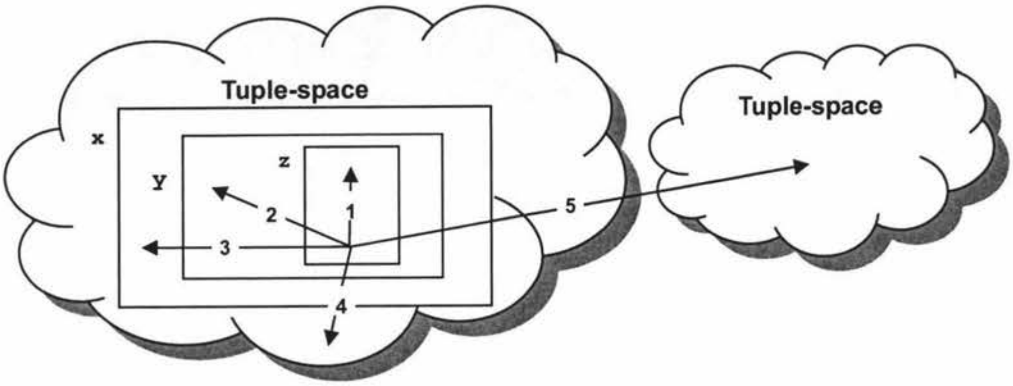


Figure18 – Process order of retrieval operations under nested transactions

When read, the found tuple is locked with a "`READ_MODE`" lock and the reference to it is added to the list recording the tuples read by the provided nested transaction. Note that a nested transaction contains the same set of lists as single-level transactions described in the previous chapter. Such a tuple may be read in any other transactions to which the tuple is visible, but cannot be taken in another transaction except itself or its child transactions. When a child transaction wants to access (i.e. read or take the tuple) a tuple locked under "`READ_MODE`", the lock will be "passed" (the transaction Id contained by the lock's Id property will be changed from the parent transaction's Id to the child transaction's Id) from the parent to the child transaction and the reference to the locked tuple will be shifted from the list recording the read tuples in the parent transaction to the child transaction's associate list. If the matching tuple is found from an alternative space, it will be recorded in the special list in the same way as for single-level transactions(See Chapter 5).

- `tk(...)`: A take matches like a read operation with the same template. When taken, the reference to the matching tuple is added to the list recording tuples taken by the provided nested transaction. Such a tuple can not be read or taken by any other transaction (including its own operations or its child transactions'). When a matching tuple is found, the tuple will be locked the same way as by a single-level transaction. Tuples locked under "`TAKE_MODE`" are not visible to any operations including operations of its own or its children's. If the matching tuple found is already locked under "READ_MODE", then the transaction needs to check the current holder of the read lock. If the lock is held by one of its parent transactions, then the provided transaction can acquire the lock and change the lock's mode to "TAKE_MODE", remove the reference to the tuple from the list maintained by the parent transaction and add the reference to the associated list inside the current transaction. The special list is also used in nested transactions to keep record of the matching tuples found from alternative spaces in the same way as single-level transactions.

Nested transaction uses the same constants for its state values as single-level transactions (See Section 5.4 for details).

6.6 Completing a Nested Transaction

A nested transaction can be completed with the code shown as below:

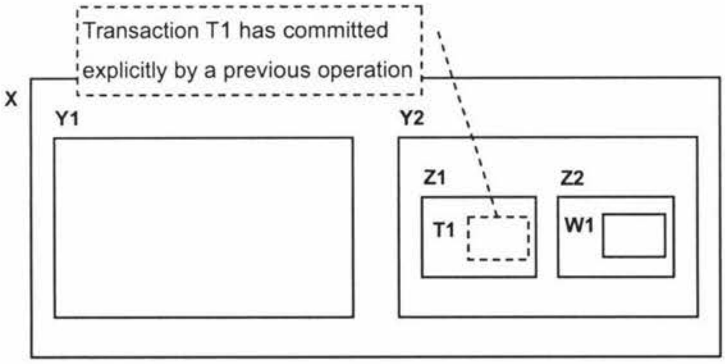
```
......
try {
    ts.commitTransaction(trx);
    ts.abortTransaction(trx1);
}catch(TransactionNotExistException e) {
    System.out.println(e);
}catch(TransactionCannotCommitException e) {
    System.out.println(e);
}catch(TransactionCannotAbortException e) {
    System.out.println(e);
}catch(TransactionException e) {
    System.out.println(e);}
```

Code Sample10 – Committing/Aborting nested transactions

The commitTransaction() and abortTransaction() operations are both one parameter methods, the only parameter needed is the NestableTransactionProxy object. Both methods have to be called within a try clause since exceptions may happen.

If a nested transaction is asked to be committed, the following steps are performed:

1. Check the state of the nested transaction. If the state of the transaction is `UPDATING`, then wait until the update process is completed. If the state is `COMMITTING`, or `COMMITTED`, or `ABORTING`, or `ABORTED`, this means that the nested transaction is already completed or under the completion process, so no more completion operation can be performed and an `InvalidTransactionState` exception will be thrown.
2. When the transaction's state is `ACTIVE`, the commit operation will lock the transaction by setting its state as `COMMITTING`, so no any new operation can be added under the transaction. If the transaction has any child transactions, the commit method for the child transactions are also called. Likewise, the commit methods for any grand child transactions are also called until the lowest level child transaction is reached and committed. If any of the child transactions has already committed or aborted, then that transaction will be skipped as its actions have already been resolved. An example of the committing sequence of a nested transaction hierarchy is shown in Figure19.


Transaction T1 has committed explicitly by a previous operation
X
Y1
Y2
Z1
Z2
T1
W1

The order of the commit operations involved when committing X:

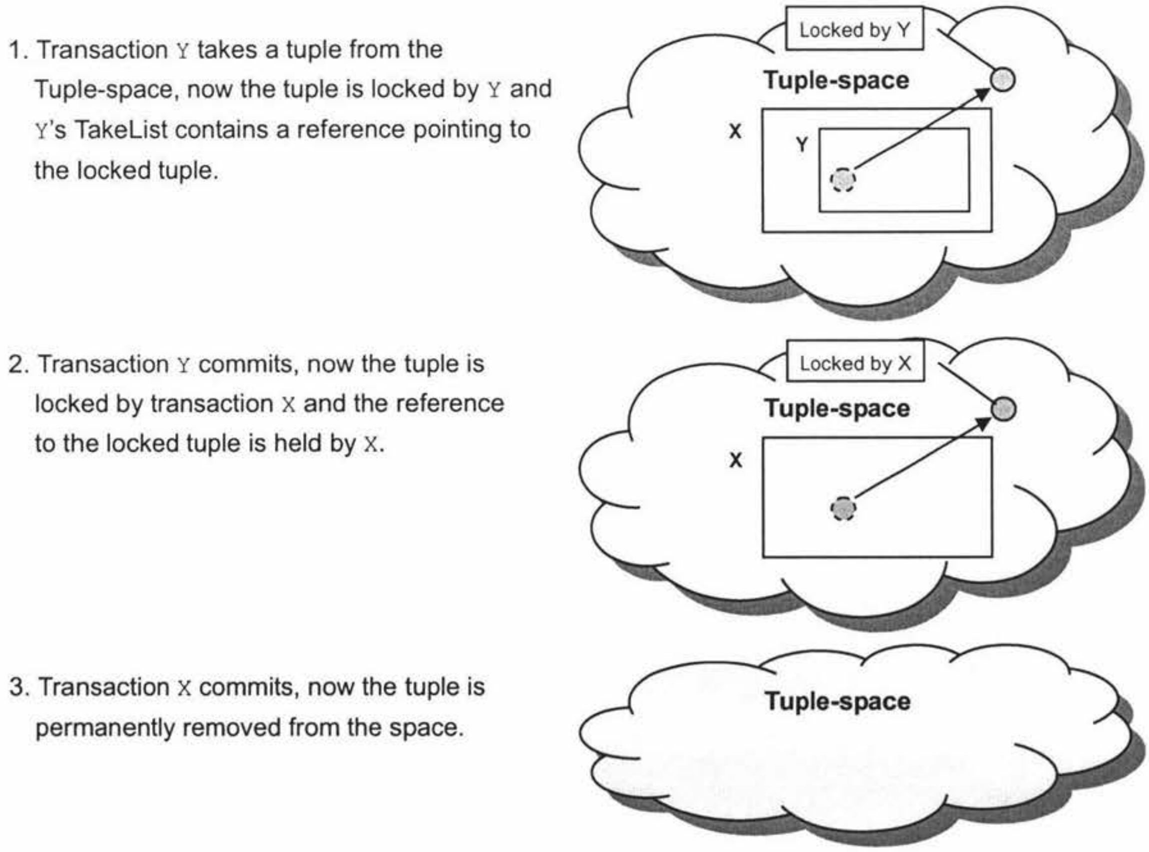
(1) Y1
(2) Z1
(3) W1
(4) Z2
(5) Y2
(6) X

Figure19 – Committing sequence of nested transactions

3. For each nested child transaction, go through the list containing the references to the tuples read from its parent transactions (if there are any) and the primary Tuple-space under the transaction, transfer the references to the relevant list in the direct parent transaction (if there is any), update the lock on each of the

recorded tuples so that the tuples are now locked by the direct parent transaction. If the committing transaction is a top-level transaction, then release the "READ_MODE" locks held by the transaction on the tuples, so these locked tuples are now free to be locked by other transactions. Clear this list.

4. For each nested child transaction, go through the list containing the references to the tuples taken from the parent transactions(if there are any) and the primary Tuple-space under the transaction. If the committing transaction has parent transaction(s), transfer the recorded references to the relevant list in the direct parent transaction, update the locks on each of the tuples so that the tuples are now locked by the direct parent transaction(as shown in Figure20). If the committing transaction is a top-level transaction, then permanently remove its "TAKE_MODE" locked tuples, so these take operations committed by the top-level transaction are irrevocable. Clear this list.



1. Transaction Y takes a tuple from the Tuple-space, now the tuple is locked by Y and Y's TakeList contains a reference pointing to the locked tuple.

2. Transaction Y commits, now the tuple is locked by transaction X and the reference to the locked tuple is held by X.

3. Transaction X commits, now the tuple is permanently removed from the space.

Figure20 – Lock passing during nested transaction committing

5. Go through the special list containing the tuples taken or read from alternative Tuple-spaces under the transaction. If the nested transaction has a parent transaction, then send a message to each of the alternative spaces that contain the taken or read tuples. The message will notify the alternative spaces to reset the "TAKE_MODE" and "READ_MODE" locks that are marked with the current transaction's Id to its parent transaction's Id. And also, the message will call a method to transfer the references to these tuples to the special list maintained in the direct parent transaction. If the nested transaction has no parent transaction (i.e. this is a top-level transaction), then the primary Tuplespace server will send a message to each of the alternative spaces that contain the taken or read tuple to notify them to permanently remove the tuples that are "TAKE_MODE" locked by the current transaction and release the "READ_MODE" lock. Clear the list.
6. For each nested transaction, go through the pool containing the tuples written under the transaction. If the nested transaction has parent transaction(s), then transfer each tuple in the pool to its direct parent transaction. If the nested transaction is a top-level transaction, then write each tuple to the primary Tuple-space and these tuples become available to any other operations. The tuples that are written to parent transaction or the space have no lock on them as the transactions locked them previously are all committed. Clear the list.
7. Set the transaction state to COMMITTED.

Note: the commit process is done in a durable fashion. When the transaction manager is going through the steps to commit the transaction, any changes it makes to the outside of its boundary (i.e. parent transactions and spaces) are recorded in a durable container. Therefore, if an exception happens at a certain point during the committing process, the transaction manager can rollback all the changes that have been made till that point. If any exception happens, the abort method is called.

If a nested transaction is asked to abort, the following steps need to be performed:

1. Check the state of the nested transaction. If the state of the transaction is UPDATING, then wait until the update process is completed. If the state is COMMITTING, or COMMITTED, or ABORTING, or ABORTED, this means that the nested transaction is already completed or under the completion process, so no more completion operation can be performed and an InvalidTransactionState exception will be thrown.
2. When the nested transaction's state is ACTIVE, the abort operation will lock the transaction by setting its state as ABORTING, so no any new operation can be added to the transaction. If the transaction has any child transactions, the abort method for the child transactions are also called. Likewise, the abort methods for any grand child transactions are also called until the lowest level child transaction is reached and aborted. If any of the child transactions has already committed or aborted, then that transaction will be skipped as its actions have already been aborted or committed to its direct parent transaction which is going to be aborted.
3. For each of the nested child transactions, go through its list recording the tuples read from its parent transactions and the primary Tuple-space under the transaction; discard any "READ_MODE" locks held by the transaction. Thus, any tuple read by the aborted transaction is set back to normal status as if the read operation has never happened. Clear this list.
4. For each of the nested child transactions, go through its list recording the tuples taken from its parent transactions and the primary Tuple-space under the transaction; discard any "TAKE_MODE" locks held by the transaction. Thus, any tuple "taken" by the aborted transaction is set back to normal status as if the take operation has never happened. Clear this list.
5. Go through the special list containing the tuples taken or read from alternative spaces under the transaction; send a message to notify each of the alternative spaces to discard the "TAKE_MODE" and "READ_MODE" locks set by the aborting transaction. Clear the list.
6. Clear the list containing the tuples written inside the transaction.
7. Set the transaction state to ABORTTED.

Note: the abort process is also done in a durable fashion. When the transaction manager is going through the steps to abort the transaction, any operations aborted are recorded in a durable container. Therefore, if an exception happens at a certain point during the aborting process, the transaction manager can try to abort the transaction again and carry on from that failure point.

As it's descried in this chapter, nested transactions involve much more work than single-level transactions. However, nested transactions do offer a number of advantages over single-level transactions:

- Nested transactions permit simple and safe composition of transactions that may execute concurrently. So nested transactions enhance system design and modularity.
- Nested transactions' object locking mechanism can help solve the concurrency problems between operations within transactions in a distributed system environment.
- Nested transactions can help protect parts of a transaction from failure in other parts, because the success or failure of each nested transaction is independent of the success of its siblings. Depending on the application's consistency requirements, a parent transaction can require retrying a failed child transaction, or try to achieve the same result in another way, or just simply ignore the failure. The failure isolation feature of nested transactions suggests that most remote actions should be considered to be performed under nested transactions.

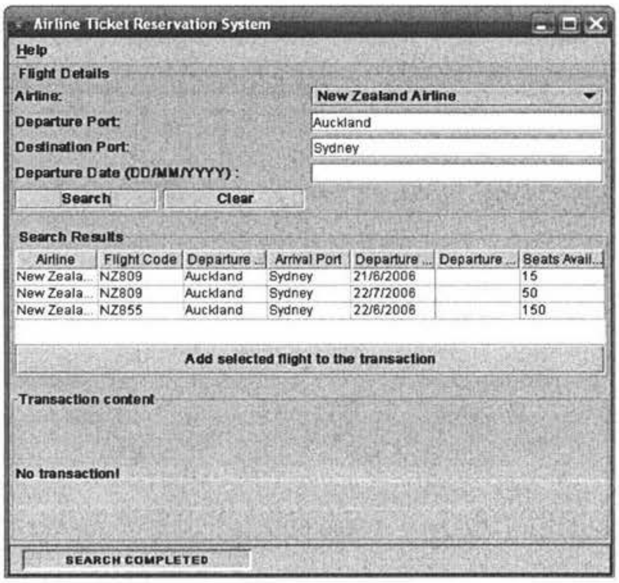
7. System Testing

This chapter is aimed to test various features and functionalities of the Tuplespace system that have been presented in the previous chapters. This chapter also includes a number of testing results representing the performance of some functions provided by the system under various circumstances. The tests are performed by developing some small programs based on the present Tuplespace system. The Tuplespace system is also tested on different platforms, e.g. UNIX, MS Windows XP.

7.1 Ticket Reservation Application

This Ticket Reservation Application is developed to test the three primary Tuplespace operations (i.e. `out(...)`, `rd(...)` and `tk(...)`), single-level transaction controls and Tuplespace server cooperation. As the purpose of this application is just to test the functionalities of the proposed Tuplespace system, it is a simple application without complicated functions. This application is designed for travel agencies to make air flight ticket reservations. The ticket information is stored as tuples in different servers that are placed at different locations. Through this application, the agent can access the flight ticket information on any of these servers and make reservation for clients.

The following screenshots demonstrate the process of a basic ticket reservation scenario:


Airline Ticket Reservation System
Help

Flight Details

Airline:	New Zealand Airline
Departure Port:	Auckland
Destination Port:	Sydney
Departure Date (DD/MM/YYYY) :	

Search Results

Airline	Flight Code	Departure ...	Arrival Port	Departure ...	Departure ...	Seats Avail...
New Zeala...	NZ809	Auckland	Sydney	21/6/2006		15
New Zeala...	NZ809	Auckland	Sydney	22/7/2006		50
New Zeala...	NZ855	Auckland	Sydney	22/6/2006		150

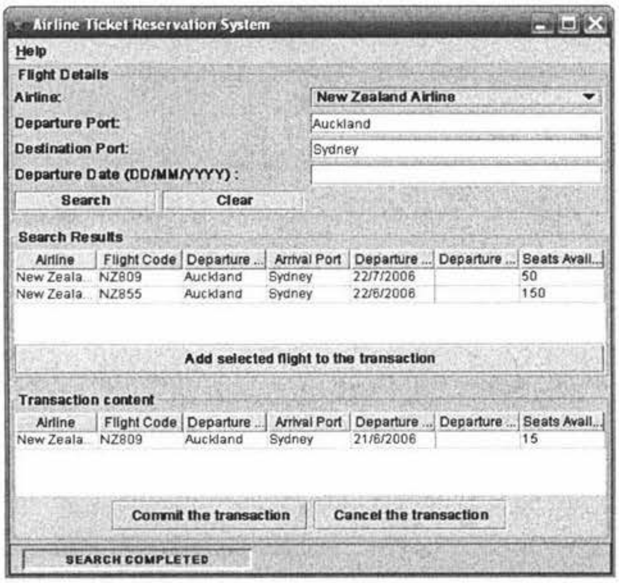
Add selected flight to the transaction

Transaction content

No transaction!

SEARCH COMPLETED

Step (1): Suppose the client wants to reserve a ticket for a New Zealand Airline flight from Auckland to Sydney on 21/06/2006. If the user leaves the "Departure Date" field blank, then the system will return any flight meeting the search criteria. The search result is shown in the table


Airline Ticket Reservation System
Help

Flight Details

Airline:	New Zealand Airline
Departure Port:	Auckland
Destination Port:	Sydney
Departure Date (DD/MM/YYYY) :	

Search Results

Airline	Flight Code	Departure ...	Arrival Port	Departure ...	Departure ...	Seats Avail...
New Zeala...	NZ809	Auckland	Sydney	22/7/2006		50
New Zeala...	NZ855	Auckland	Sydney	22/6/2006		150

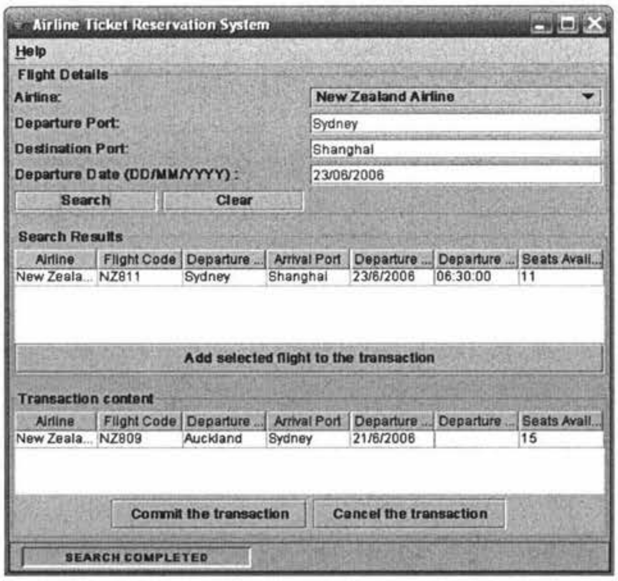
Add selected flight to the transaction

Transaction content

Airline	Flight Code	Departure ...	Arrival Port	Departure ...	Departure ...	Seats Avail...
New Zeala...	NZ809	Auckland	Sydney	21/6/2006		15

SEARCH COMPLETED

Step (2): The user selects the flight meets the client's request, then click on the "Add selected flight to the transaction" button to add the flight to the transaction whose contents are shown in the bottom table.


Airline Ticket Reservation System
Help

Flight Details

Airline:	New Zealand Airline
Departure Port:	Sydney
Destination Port:	Shanghai
Departure Date (DD/MM/YYYY) :	23/06/2006

Search Results

Airline	Flight Code	Departure ...	Arrival Port	Departure ...	Departure ...	Seats Avail...
New Zeala...	NZ811	Sydney	Shanghai	23/6/2006	06:30:00	11

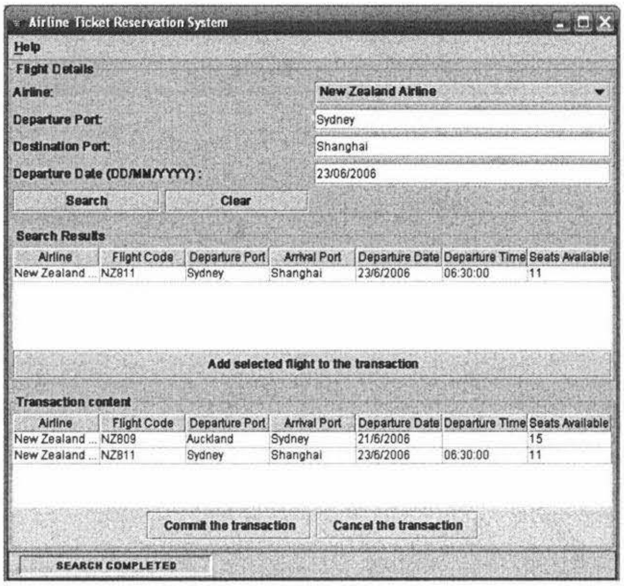
Add selected flight to the transaction

Transaction content

Airline	Flight Code	Departure ...	Arrival Port	Departure ...	Departure ...	Seats Avail...
New Zeala...	NZ809	Auckland	Sydney	21/6/2006		15

SEARCH COMPLETED

Step (3): If the client wants to carry on another flight on 23/06/2006 from Sydney to Shanghai, the agent then search again, and the new result is shown in the table in the middle of the window.


Airline Ticket Reservation System
Help
Search
Clear
Add selected flight to the transaction
Commit the transaction
Cancel the transaction

Flight Details

Airline: New Zealand Airline

Departure Port: Sydney

Destination Port: Shanghai

Departure Date (DD/MM/YYYY) : 23/06/2006

Search Results

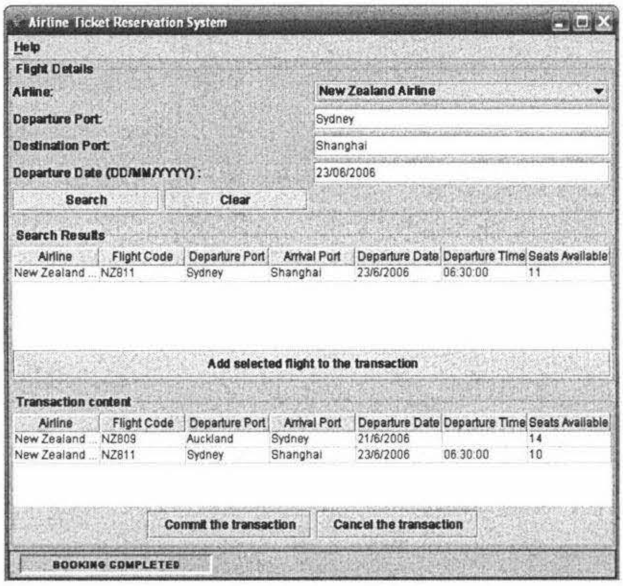
Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
New Zealand ...	NZ811	Sydney	Shanghai	23/6/2006	06:30:00	11

Transaction content

Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
New Zealand ...	NZ809	Auckland	Sydney	21/6/2006		15
New Zealand ...	NZ811	Sydney	Shanghai	23/6/2006	06:30:00	11

SEARCH COMPLETED

Step (4): Click on the "Add selected flight to the transaction" button again to add the new flight to the transaction whose updated contents are shown in the bottom table.


Airline Ticket Reservation System
Help
Search
Clear
Add selected flight to the transaction
Commit the transaction
Cancel the transaction

Flight Details

Airline: New Zealand Airline

Departure Port: Sydney

Destination Port: Shanghai

Departure Date (DD/MM/YYYY) : 23/06/2006

Search Results

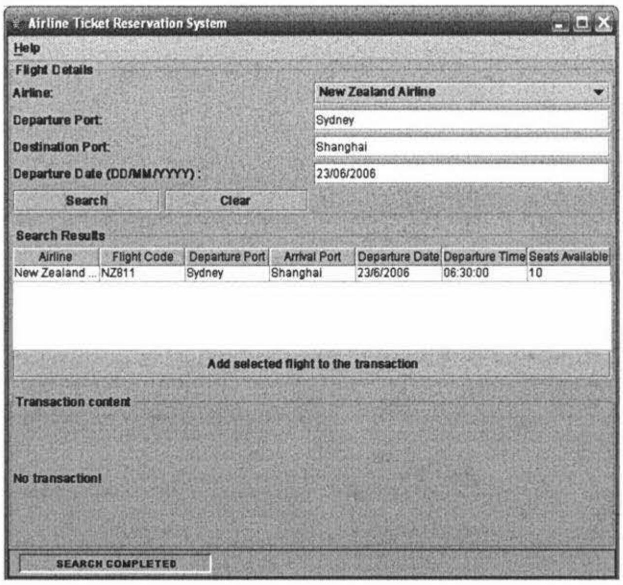
Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
New Zealand ...	NZ811	Sydney	Shanghai	23/6/2006	06:30:00	11

Transaction content

Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
New Zealand ...	NZ809	Auckland	Sydney	21/6/2006		14
New Zealand ...	NZ811	Sydney	Shanghai	23/6/2006	06:30:00	10

BOOKING COMPLETED

Step (5): When the transaction contains all the flights the client requires, then the user clicks on the "Commit the transaction" button. As we can see, the "Seats Available" numbers shown in the "Transaction content" table are reduced by one.


Airline Ticket Reservation System
Help
Search
Clear
Add selected flight to the transaction

Flight Details

Airline: New Zealand Airline

Departure Port: Sydney

Destination Port: Shanghai

Departure Date (DD/MM/YYYY) : 23/06/2006

Search Results

Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
New Zealand ...	NZ811	Sydney	Shanghai	23/6/2006	06:30:00	10

Transaction content

No transaction!

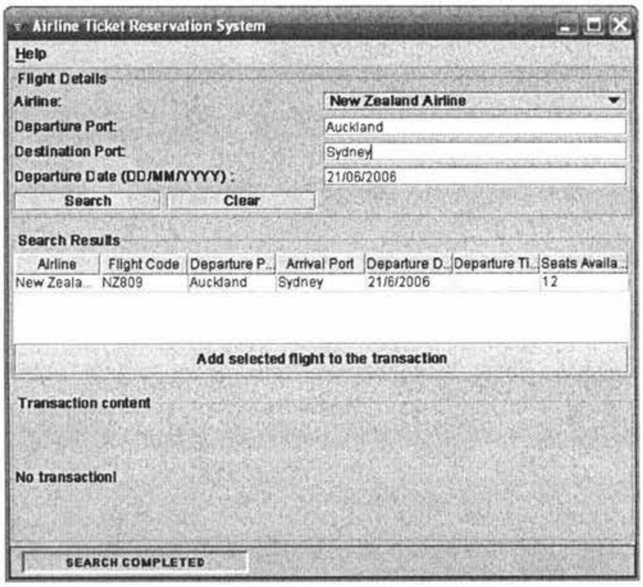
SEARCH COMPLETED

Step (6): If the agent searches for the New Zealand Airline flight from Sydney to Shanghai again, he will find out that the Seats Available number is one less than before now.

Figure21 – Demonstration of a basic ticket reservation process (1)

In Figure21, the `out(…)`, `rd(…)`, `tk(…)` and tuple fields retrieval operations were tested to retrieve desired flight information from the space(s) and write updated tuples back to the space(s). When the user presses on the "Commit the Transaction" button, the single-level transaction management functions are invoked, and the effect of the transaction become permanent in the space(s).

In the following scenario, the cooperation among the Tuplespace servers and some more issues about transactions are tested.



Airline Ticket Reservation System
Help
Flight Details

Airline: New Zealand Airline
Departure Port: Auckland
Destination Port: Sydney
Departure Date (DD/MM/YYYY) : 21/06/2006

Search
Clear
Search Results

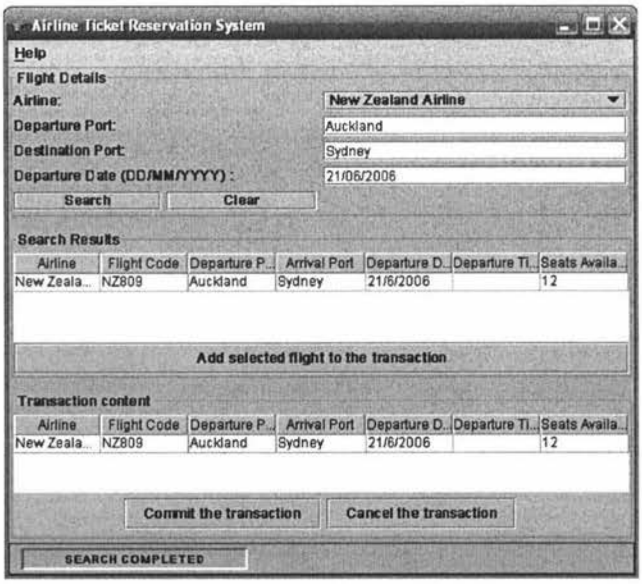
Airline	Flight Code	Departure P...	Arrival Port	Departure D...	Departure Ti...	Seats Availa...
New Zeala...	NZ809	Auckland	Sydney	21/6/2006		12

Add selected flight to the transaction
Transaction content

No transaction!

SEARCH COMPLETED

Step (1): Suppose the client wants to reserve a ticket for a New Zealand Airline flight from Auckland to Sydney on 21/06/2006. The search result is shown in the table.



Airline Ticket Reservation System
Help
Flight Details

Airline: New Zealand Airline
Departure Port: Auckland
Destination Port: Sydney
Departure Date (DD/MM/YYYY) : 21/06/2006

Search
Clear
Search Results

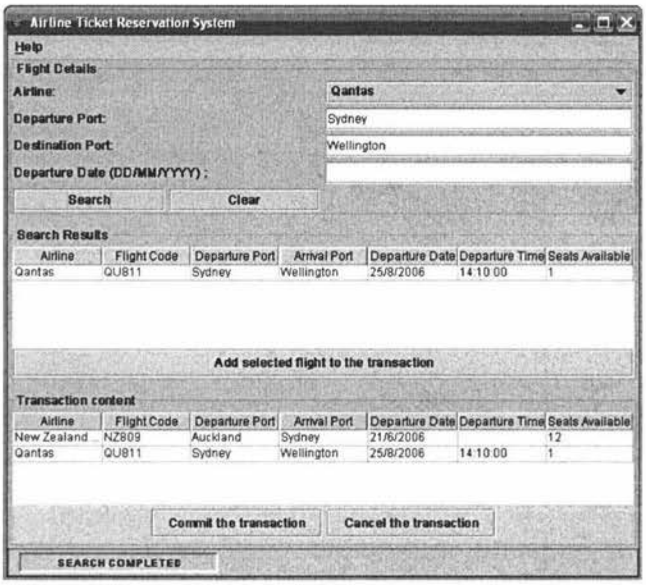
Airline	Flight Code	Departure P...	Arrival Port	Departure D...	Departure Ti...	Seats Availa...
New Zeala...	NZ809	Auckland	Sydney	21/6/2006		12

Add selected flight to the transaction
Transaction content

Airline	Flight Code	Departure P...	Arrival Port	Departure D...	Departure Ti...	Seats Availa...
New Zeala...	NZ809	Auckland	Sydney	21/6/2006		12

SEARCH COMPLETED

Step (2): Click on the "Add selected flight to the transaction" button to add the flight to the transaction whose content is shown in the bottom table.


Airline Ticket Reservation System
Help
Search
Clear
Add selected flight to the transaction
Commit the transaction
Cancel the transaction

Flight Details

Airline: Qantas

Departure Port: Sydney

Destination Port: Wellington

Departure Date (DD/MM/YYYY) :

Search Results

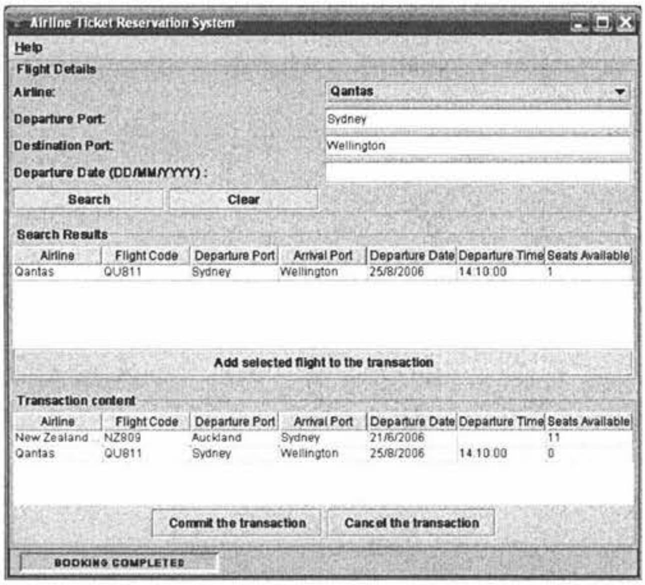
Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
Qantas	QU811	Sydney	Wellington	25/8/2006	14:10:00	1

Transaction content

Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
New Zealand ...	NZ809	Auckland	Sydney	21/6/2006		12
Qantas	QU811	Sydney	Wellington	25/8/2006	14:10:00	1

SEARCH COMPLETED

Step (3): Add another flight of QANTAS Airline to the transaction. Actually, the flight information of QANTAS Airline is retrieved from another Tuplespace server. As it is shown, the QANTAS flight has only 1 seat left.


Airline Ticket Reservation System
Help
Search
Clear
Add selected flight to the transaction
Commit the transaction
Cancel the transaction

Flight Details

Airline: Qantas

Departure Port: Sydney

Destination Port: Wellington

Departure Date (DD/MM/YYYY) :

Search Results

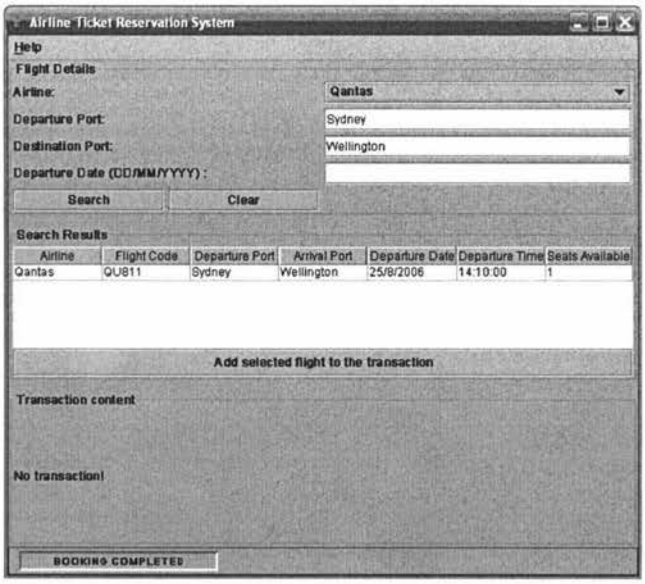
Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
Qantas	QU811	Sydney	Wellington	25/8/2006	14:10:00	1

Transaction content

Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
New Zealand ...	NZ809	Auckland	Sydney	21/6/2006		11
Qantas	QU811	Sydney	Wellington	25/8/2006	14:10:00	0

BOOKING COMPLETED

Step (4): When the user has got all the flights he needs as shown in Step(3), he clicks on the "Commit the Transaction" button to commit the transaction. And the contents in the transaction table are updated. Now there are 11 seats left in the New Zealand Airline flight and no seat left in the QANTAS Airline flight.


Airline Ticket Reservation System
Help
Search
Clear
Add selected flight to the transaction

Flight Details

Airline: Qantas

Departure Port: Sydney

Destination Port: Wellington

Departure Date (DD/MM/YYYY) :

Search Results

Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
Qantas	QU811	Sydney	Wellington	25/8/2006	14:10:00	1

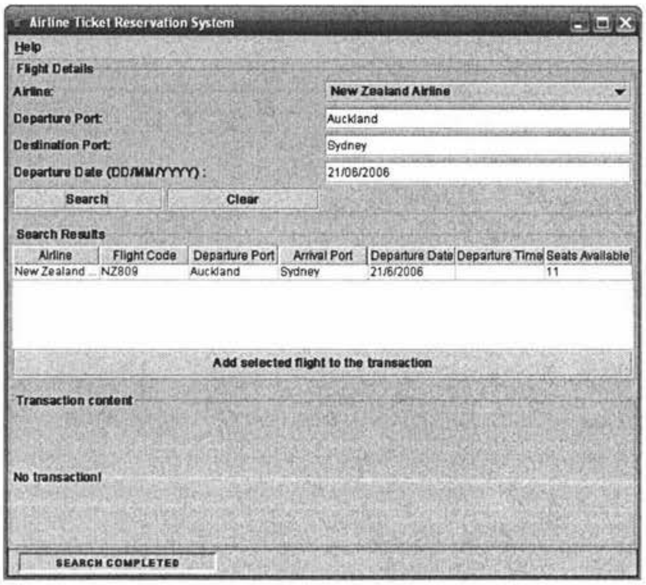
Transaction content

No transaction!

BOOKING COMPLETED

Step (5): If the user click on the "Commit the transaction" button again to make the same reservation one more time, the system will abort the transaction as there is no seat left in the QANTAS flight. Thus, if a change has been made to the New Zealand Airline flight, the change has to be cancelled as the reservation can not be made successfully as a whole transaction.

Step (6): If the user checks the information of the New Zealand Airline flight again, he will see that the number of seats available is 11 not 10 which means that the second reservation made in Step(5) didn't happen.


Airline Ticket Reservation System
Help
Search
Clear
Add selected flight to the transaction

Flight Details

Airline: New Zealand Airline

Departure Port: Auckland

Destination Port: Sydney

Departure Date (DD/MM/YYYY) : 21/06/2006

Search Results

Airline	Flight Code	Departure Port	Arrival Port	Departure Date	Departure Time	Seats Available
New Zealand ...	NZ809	Auckland	Sydney	21/6/2006		11

Transaction content

No transaction!

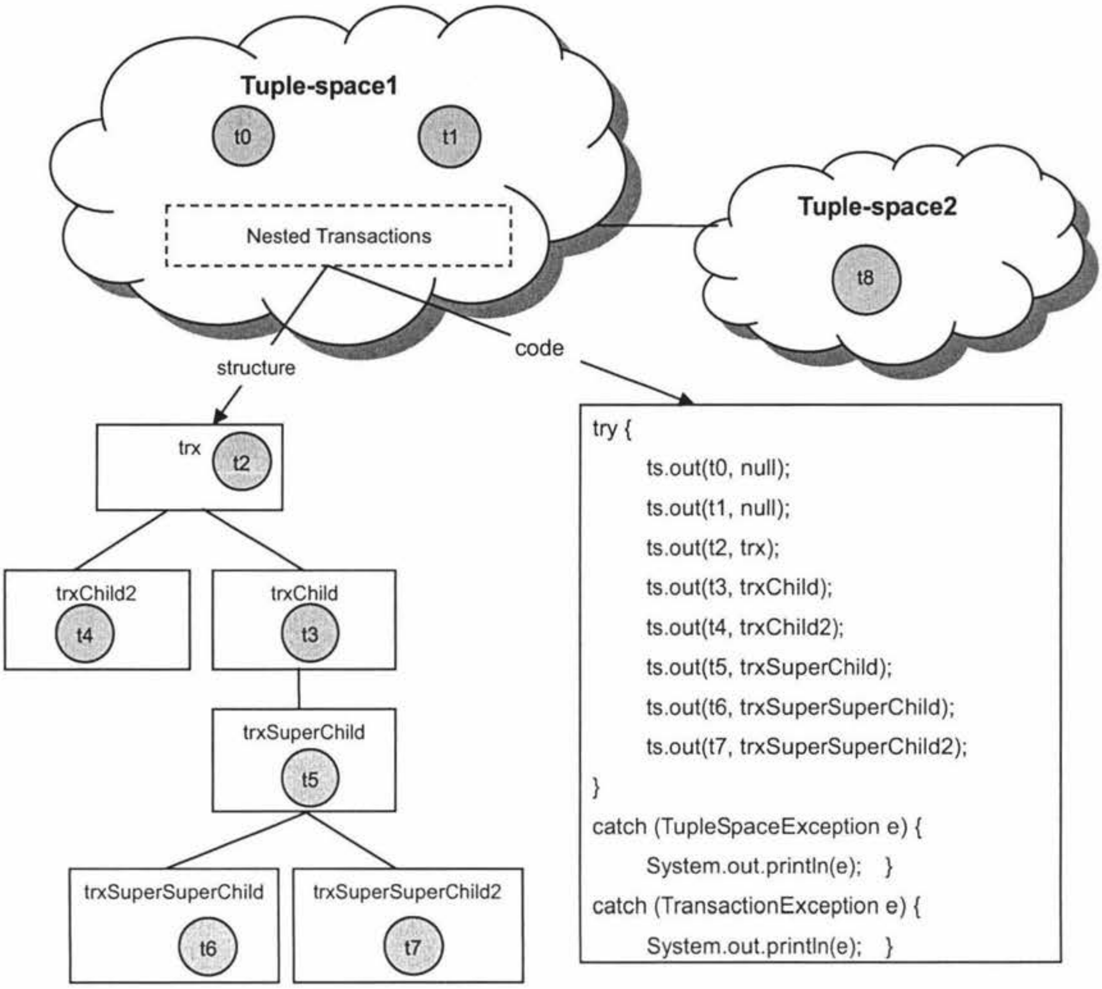
SEARCH COMPLETED

Figure22 – Demonstration of a basic ticket reservation process (2)

Transaction abortion is tested in the second demonstration. Though the tuples representing flight information are from different spaces, the cooperation between Tuplespace servers is invisible to the client.

7.2 Nested Transaction Testing

In this section, programs to test various features of Nested Transactions are discussed. At the beginning of this test program, it is going to set up a nested transaction hierarchy and output some tuples to the Tuplespace and each of the nested transactions. Now, the status of the Tuplespace is shown as Figure23 and fields of each tuple are shown in Figure24.



```
try {
        ts.out(t0, null);
        ts.out(t1, null);
        ts.out(t2, trx);
        ts.out(t3, trxChild);
        ts.out(t4, trxChild2);
        ts.out(t5, trxSuperChild);
        ts.out(t6, trxSuperSuperChild);
        ts.out(t7, trxSuperSuperChild2);
}
catch (TupleSpaceException e) {
        System.out.println(e);  }
catch (TransactionException e) {
        System.out.println(e);  }
```

Figure23 – Status of the test Tuplespace system

t0

Field	Type	Value
0	java.lang.String	"Non"
1	java.lang.Integer	0

t1

Field	Type	Value
0	java.lang.String	"Non2"
1	java.lang.Integer	0

t2

Field	Type	Value
0	java.lang.String	"Trx"
1	java.lang.Integer	1

t3

Field	Type	Value
0	java.lang.String	"Child"
1	java.lang.Integer	2

t4

Field	Type	Value
0	java.lang.String	"Child2"
1	java.lang.Integer	2

t5

Field	Type	Value
0	java.lang.String	"SuperChild"
1	java.lang.Integer	3

t6

Field	Type	Value
0	java.lang.String	"Super Super Child"
1	java.lang.Integer	4

t7

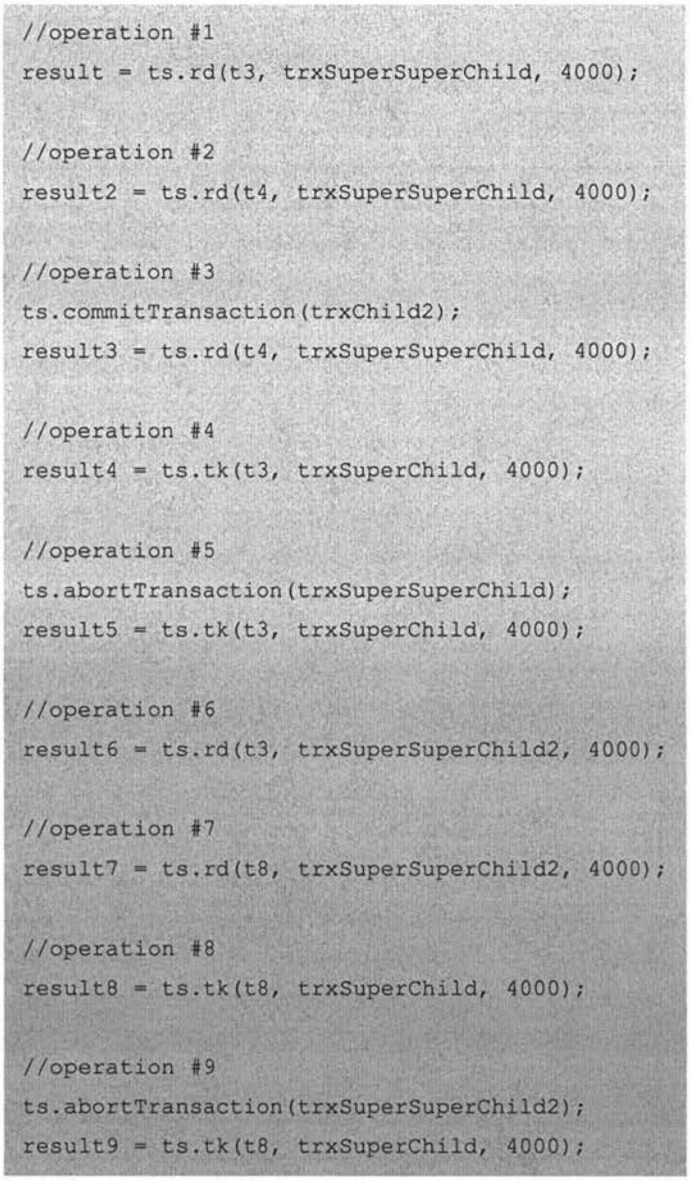
Field	Type	Value
0	java.lang.String	"Super Super Child2"
1	java.lang.Integer	4

t8

Field	Type	Value
0	java.lang.String	"Remote"
1	java.lang.Integer	5

Figure24 – Contents of the test tuples

After the Tuplespace is set up, now the test program will execute the following operations one by one.



```
//operation #1
result = ts.rd(t3, trxSuperSuperChild, 4000);

//operation #2
result2 = ts.rd(t4, trxSuperSuperChild, 4000);

//operation #3
ts.commitTransaction(trxChild2);
result3 = ts.rd(t4, trxSuperSuperChild, 4000);

//operation #4
result4 = ts.tk(t3, trxSuperChild, 4000);

//operation #5
ts.abortTransaction(trxSuperSuperChild);
result5 = ts.tk(t3, trxSuperChild, 4000);

//operation #6
result6 = ts.rd(t3, trxSuperSuperChild2, 4000);

//operation #7
result7 = ts.rd(t8, trxSuperSuperChild2, 4000);

//operation #8
result8 = ts.tk(t8, trxSuperChild, 4000);

//operation #9
ts.abortTransaction(trxSuperSuperChild2);
result9 = ts.tk(t8, trxSuperChild, 4000);
```

```
//operation #10
result10 = ts.rd(t9, trxSuperChild, 10000);

//operation #11
ts.commitTransaction(trxChild);
result11 = ts.rd(t5, null, 4000);

//operation #12
ts.commitTransaction(trx);
result12 = ts.rd(t4, null, 2000);

//operation #13
result13 = ts.rd(t3, null, 2000);
```

The result of the above operations is shown and explained in the following table:

#	Result	Comment
1	`<Child, 2>`	`trxSuperSuperChild`'s great grand parent transaction `trxChild` has the matching tuple and therefore it is returned.
2	`null`	No matching tuple found. As `trxChild2` is not a parent transaction of `trxSuperSuperChild`, the tuple stored inside `trxChild2` is not visible to `trxSuperSuperChild`.
3	`<Child, 2>`	When `trxChild2` commits, its tuple is written to its direct parent transaction which is `trx` in this case. As `trx` is a parent transaction of `trxSuperSuperChild`, `trxSuperSuperChild` this time is able to find a matching tuple in `trx`.
4	`null`	`trxSuperChild` failed to take the tuple currently stored in its direct parent transaction `trxChild`, because `t3` has already been "READ_MODE" locked by `trxSuperChild`'s child transaction `trxSuperSuperChild` at operation #1 and the child transaction's lock always wins.
5	`<Child, 2>`	When `trxSuperSuperChild` is aborted, all the locks held by it are discarded. Thus, `trxSuperChild` is now able to take the tuple from `trxChild`.

6	null	As t3 has already been taken by trxSuperChild in last operation, t3 is not available to any transactions including the child transactions. This operation also shows that trxSuperSuperChild2 is working separately from trxSuperSuperChild. In other words, the abortion of trsSuperSuperChild doesn't affect the lifetime of trxSuperSuperChild2.
7	<Remote, 5>	trxSuperSuperChild2 reads the tuple from Tuple-space2.
8	null	trxSuperChild cannot take the tuple from Tuple-space2 as the tuple is ready "READ_LOCK" by its child transaction trxSuperSuperChild2 in last operation.
9	<Remote, 5>	As trxSuperSuperChild2 aborts, all the locks (including the locks in Tuple-space2) held by it are discarded. Thus, trxSuperChild is now able to take the tuple from Tuple-space2.
10	<New, 5>	trxSuperChild is trying to read a tuple which is currently not available in neither of the Tuple-spaces. It is defined in the method that the operation is going to wait for 10 seconds for the matching tuple to turn up. In the meantime, we start another client program which writes a matching tuple to Tuple-space1. The result shows that the read operation is able to detect and read the newly inserted tuple while it is waiting.
11	null	The operation demonstrates that the tuples written within a nested transaction is not available until the top-level transaction commits successfully.
12	<Child2, 2>	As the top-level transaction trx commits successfully, all of the tuples written within its scope are visible to the outside world.
13	null	As t3 has already been taken by operation #5 within the transaction, it does not get written to the space.

(Note: If a matching tuple is returned, its content will be shown, otherwise null is shown)

In this test program, various features of nested transactions are tested, such as concurrency control, lock management, state restoration, alternative Tuple-space access, timeout waiting, and synchronization control. This simple program also demonstrates that nested transaction can offer a number of extra features over single-level transactions.

Another two programs are presented to test the reliability of nested transactions under extreme conditions.

Test1:

Start 10 threads simultaneously, each thread writes 10,000 tuples to the same Tuple-space. The tuple to be written is of a complicated structure, which consists of 10 `String` type fields, 10 `SuperDate`(inherited from `java.util.Date`) type fields and an `Integer` type field. In the meantime, start another 10 threads to retrieve tuples from the Tuple-space and each retrieval thread takes 10,000 tuples from the Tuple-space under different levels of nested transactions. When these 20 threads finish, there are no more tuples left in the space. This test proves that the presented Tuplespace system is able to maintain the ACID properties of nested transactions under concurrent situations with big volumes of input and output.

Test2:

Write 100,000 tuples to a Tuple-space under different nested transactions of a nested transactional hierarchy. After the 100,000 tuples are stored in the space, read and take the 100,000 tuples from the Tuple-space under different nested transactions. This test proves that nested transactions are able to maintain ACID properties and locking rules under the boundary conditions.

7.3 Performance Testing

We have implemented a number of performance tests under different circumstances in a networked environment. The results of the tests show that the performance of the system could be affected by various factors, such as the size and complexity of the tuple, workload of the Tuplespace server and network latency. The performance tests in this section were performed on the Helix Computer at Albany Campus, Massey University.

7.3.1 Write tuples

In order to test how the performance of the `out(...)` operation is affected by the size and complexity of the tuple, each of the tuples in the following table was written to a

Tuplespace, and the result of the performance is shown in Figure26. As we can see from the following table, `tupleSimple` is a very basic and simple tuple; `tupleLong` has the same structure as `tupleSimple` but one of its fields is bigger; `tupleComplex` has similar content size as `tupleSimple` but it has more fields.

Tuple	Fields	
tupleSimple	Type: java.lang.String	Value: "abcdefghijklmnopqrstuvwxyz"
	Type: java.lang.Integer	Value: 7777
tupleLong	Type: java.lang.String	Value: "abcdefghijklmnopqrstuvwxyz" + "abcdefghijklmnopqrstuvwxyz" + "abcdefghijklmnopqrstuvwxyz" + "abcdefghijklmnopqrstuvwxyz" + "abcdefghijklmnopqrstuvwxyz" + "abcdefghijklmnopqrstuvwxyz" + ... "abcdefghijklmnopqrstuvwxyz" (20 times)
	Type: java.lang.Integer	Value: 7777
tupleComplex	Type: java.lang.String	Value: "a"
	Type: java.lang.String	Value: "b"
	Type: java.lang.String	Value: "c"
	Type: java.lang.String	Value: "d"
	....	......
	Type: java.lang.String	Value: z
	Type: java.lang.Integer	Value: 7777

Figure25 – Tuples used for testing

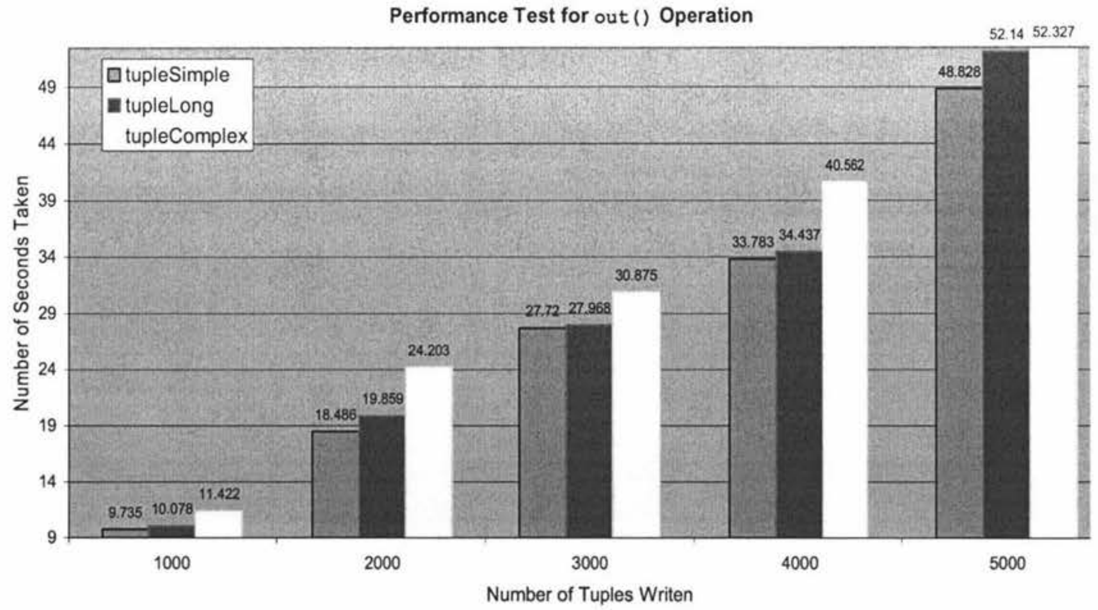


Figure26 – Performance Test for `out()` operations

Figure26 shows that the approximate performance of the out(...) operation is predictable as it is related to the complexity and structure of the tuple to be written. However, it is difficult to calculate the exact performance of this operation because the performance is also affected by the state of the server.

In order to compare the performance of the out(...) operations with/without transactional controls, the following test writes tupleSimple under three conditions: without any transaction, under a single level transaction and under a nested transaction. The result is shown in Figure27. The test result shows that the out(...) operations take longer to finish when it is done under transactional controls. The performance difference between single-level and multiple-level transactions is not stable because that this result is affected by the workload on the server when it's implementing the operation.

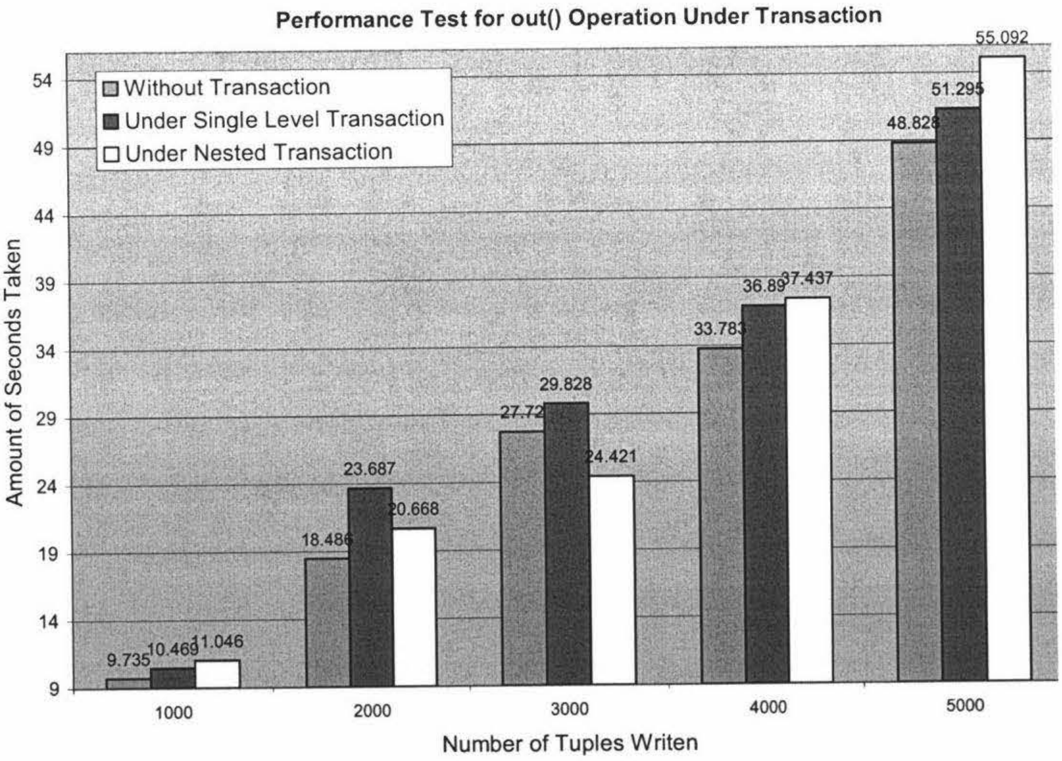


Figure27 – Performance Test for `out()` operations under (sub)transactions

Figure 28 shows the performance difference of the out(...) operations between writing to an empty Tuple-space and to a filled Tuple-space. For this test, the tupleSimple object was written 1000 times to the Tuple-space when it was containing different quantity of tuples.

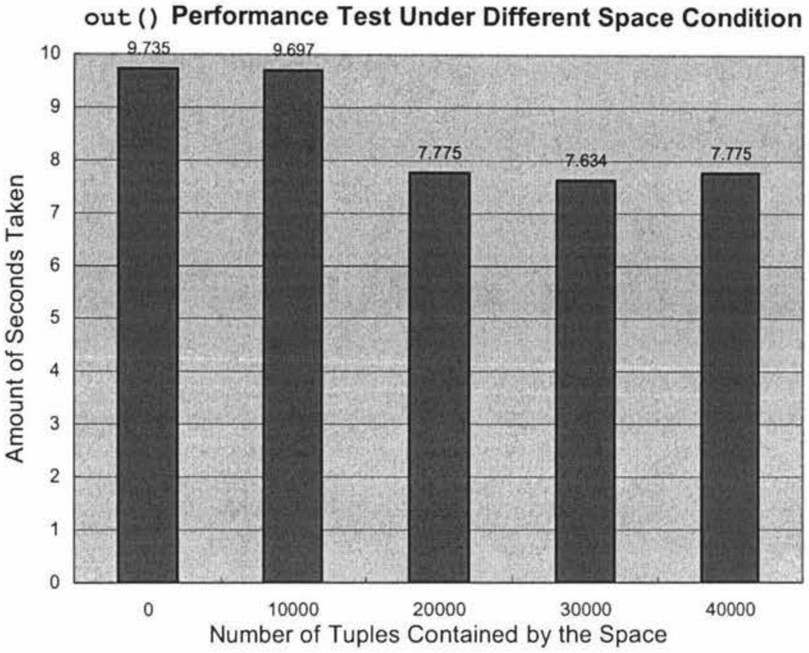


Figure28 – Performance Test for `out()` operations under different space conditions

The following test shows the performance difference of the `out(...)` operations when the Tuplespace server is serving different quantity of concurrent accesses. For this test, the `tupleSimple` object was written 1000 times to the space when the Tuplespace server was serving 0, 25, 50, 75, and 100 clients.

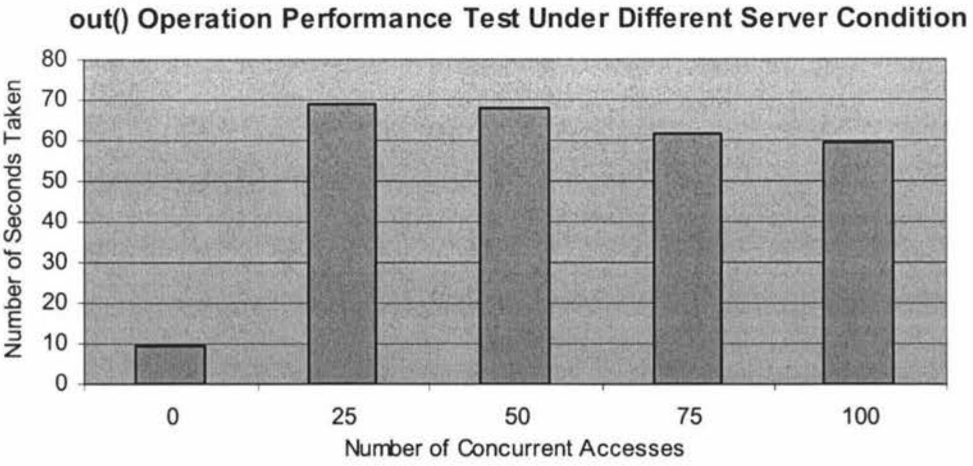


Figure29 – Performance Test for `out()` operations under different server conditions

The result of this test shows that the performance of the Tuplespace server is greatly affected when the server is serving multiple clients simultaneously. Moreover, when the server is dealing with multiple clients, its exact performance is quite unpredictable.

7.3.2 Read/Take tuples

In this section, the performance tests are implemented on the `rd(...)` operations only because the performance of `tk(...)` operations are almost identical as the `rd(...)` operations.

A program was developed to read each of the tuples in Figure25 as template from a Tuple-space to test the performance of the `rd(...)` operations. Figure30 shows the performance result when reading a matching tuple 1000 times from the Tuple-space. Although the performance of the `rd(...)` operation cannot be predicated exactly(the result also depends on the state of the server), this test proves that the performance of `rd(...)` operation is related to the structure and complexity of the tuple to be read.

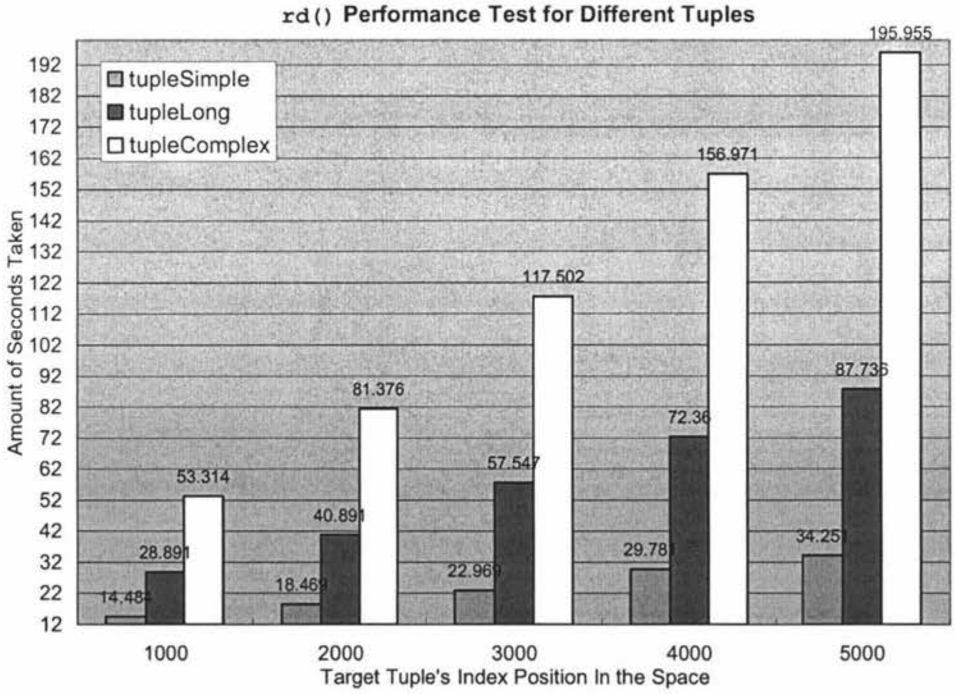


Figure30 – Performance Test for `rd()` operations

In order to compare the performance of the `rd(...)` operations with/without transactional controls, the following test reads `tupleSimple` under three conditions: without any transaction, under a single-level transaction and under a nested transaction. The result is shown in Figure 31.

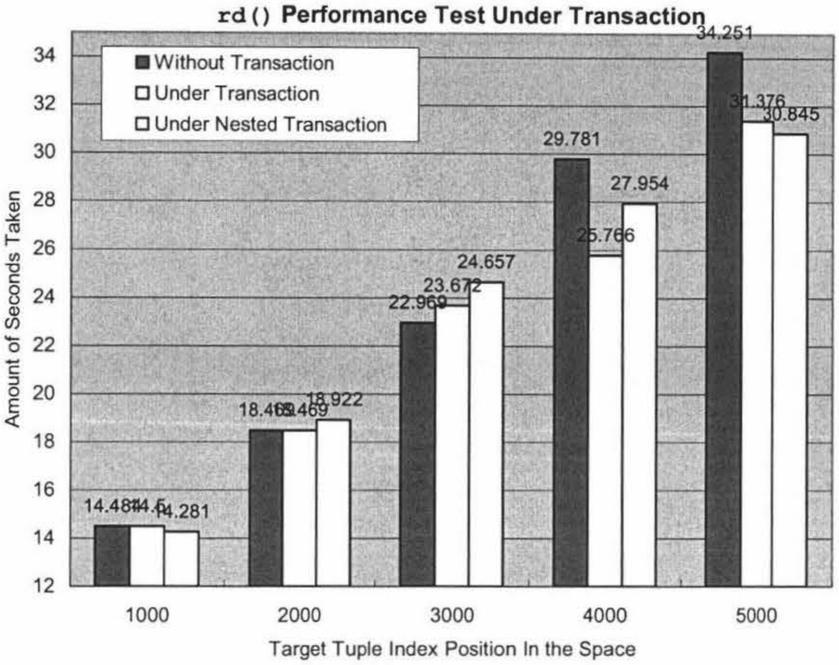


Figure31 – Performance Test for `rd()` operations under transactions

The following chart demonstrates the performance difference of the `rd(...)` operations between retrieving tuple from the primary Tuple-space and from an alternative Tuple-space. For this test, the program tried to read a tuple which is not available in the primary Tuple-space, so the primary Tuple-space redirects the request to another Tuple-space (an alternative Tuple-space) for further search. The result (Figure32) clearly shows that the performance is greatly affected by the network latency.

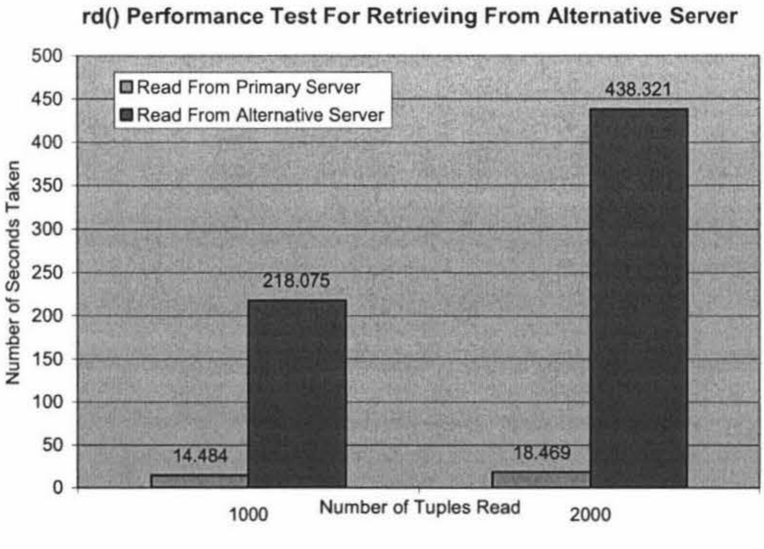


Figure32 – Performance Test for `rd()` when retrieving from alternative servers

The following test shows the performance difference of the `rd(...)` operation when the server is serving different number of concurrent accesses.

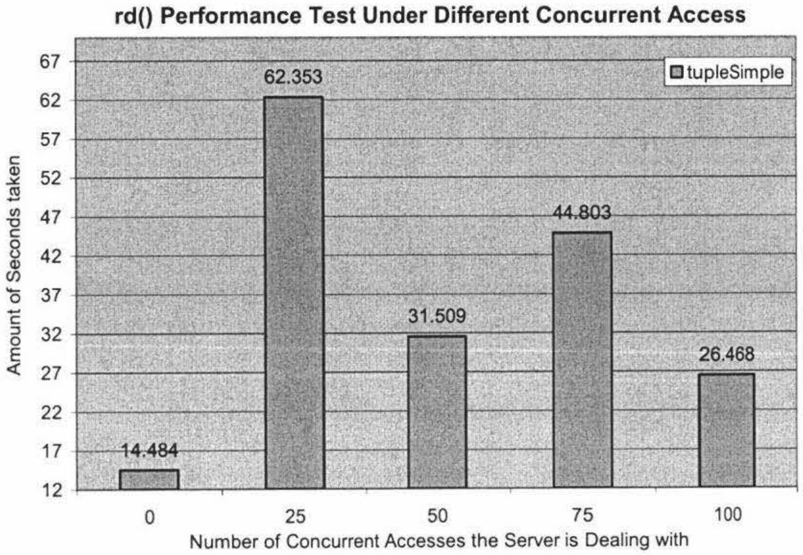


Figure33 – Performance Test for `rd()` under concurrent access situations

The tests implemented in this section demonstrate that the performance of the Tuplespace system is affected by the size and complexity of the tuples to be wrote or retrieved. However, the performance difference between implementing the operations with/without transactional controls is not significant. When the Tuplespace server is dealing with different number of concurrent accesses, the performance is greatly affected and unpredictable, because the Tuplespace server will randomly pick up tasks from a task pool and process them. Any request from clients will be placed in a task pool first. The Tuplespace server is designed to have 100 threads running concurrently to take tasks from the task pool, complete the task and then take another task from the task pool. Therefore, there is no precise indication about the performance under a concurrent situation as we don't know the position of the task in the pool and when it will be taken for process. Another point needs to be noticed from the tests is that the network latency can dramatically affect the performance of the system. When the client's request is redirected to alternative Tuplespace servers, i.e. the request is send to another computer over a network, the performance difference between different types of tuples is totally overwhelmed by the network latency.

In this chapter, we've shown tests incorporating both single-level and nested transactions into applications so that they can operate in a safe and correct manner in the presence of partial failure. We've also shown tests demonstrating the uses of each operation provided by the system and how to make use of multiple spaces in order to make the application become more truly scalable. Based on these tests, we can see that the Tuplespace model and this Tuplespace system could be a powerful option for building robust and scalable distributed applications.

8. Further Research Areas

There are number of interesting areas for further research. This chapter discusses some of such topics in turn.

8.1 Client Access Authorization and Management

In the current design of the system, as long as clients know how to connect to the Tuplespace server and comply with the communication protocols, they can write tuples into the Tuple-space and retrieve tuples out from the Tuple-space(s), and each client has equivalent access right. In other words, a client can potentially access all the tuples stored in the space(s), including the tuples containing sensitive data. It would be useful if the server could firstly validate the clients that access its Tuple-space, and control their access levels. By having access control, the tuples containing sensitive data, such as password, bank account information or personal information, can be stored securely in the space and are only accessible to those authorized clients. And also, access control ensures that the resources of the Tuplespace server are only allocated to authorized users.

8.2 Dynamic Class Loading

This Tuplespace system is designed and built based on the assumption that the client "understands" the tuple returned from the Tuplespace server including all the fields contained in the tuple. Likewise, the Tuplespace server is also assumed to be able to "fully understand" each tuple and template it receives from the clients. By "understand", we mean that each end of the system has already got the same version of the `.class` files to interpret (deserialize) the tuple objects and field objects. However, in a fully distributed system, the assumption cannot always be true. For example, client A writes `tuple1` containing a field of type `Orange` to a Tuple-space, client B wants to read a tuple which contains a field of type `Orange`, so `tuple1` is returned to client B. However, client B is holding an old version of `Orange.class`, which means that client B cannot interpret `tuple1` accurately. To solve this problem, the Tuplespace system has to have support for dynamic class loading to ensure that each node of the system has the proper `.class` files to interpret the objects.

Java platform has the capability to dynamically download Java software from any URL to a JVM running in a separate process, usually on a different physical system. For example, a JVM running from within a web browser can download the bytecodes for subclasses of `java.applet.Applet` and any other classes needed by that applet. The system on which the browser is running has likely never run this applet before, nor installed it on its disk. Once all the necessary classes have been downloaded from the server, the browser can start the execution of the applet program using the local resources of the system on which the client browser is running. Moreover, Java RMI takes advantage of this capability to download and execute classes and on systems where those classes have never been installed on disk. Therefore, this capability of Java platform can be investigated in further research to achieve dynamic class loading in the system.

Here are some suggestions about the techniques that might be involved when building dynamic class loading supports for the system in further research. One is the use of `ClassLoader` in the Java programming language. `ClassLoader` allows Java program to load classes from known location(s). Usually, a class loader is used in conjunction with an HTTP server that is serving up compiled classes for the Java platform. A second technique is the use of Java `codebase`. Java `codebase` can be defined as a source, or a place, from which to load classes into a JVM. `Codebase` is similar as CLASSPATH of a system and give directions to a Java application to find desired (potentially remote) classes. The third technique that could be used is Java `MarshalledObject`. A `MarshalledObject` contains a byte stream with the serialized representation of an object given to its constructor. The classes needed for the serialized object are annotated with a codebase URL from where the class can be loaded (if available).

8.3 Higher Efficiency and Reliability

There are many possibilities to enhance the performance and reliability of the Tuplespace system. The following list shows some points that can be investigated for efficiency and reliability improvements:

- Buffer output and input when generating and receiving messages
- Design a protocol to efficiently packetize messages and re-assemble them, including low level buffering and flow control, sequence numbering, acknowledgment and retransmission scheme.
- More efficient organization and format of the various lists that transaction requires.
- Develop lease control of transactions. Thus, the lifetime of transactions can be controlled by transaction manager, and no transaction can occupy server resources forever.
- Build a replication scheme on top of the Tuplespace system to provide multiple copy Tuple-spaces while simultaneously addressing concurrency and reliability problems.
- Develop a more efficient matching algorithm. For example, in [8], a system named RDBSpace was developed which utilizes a relational database back-end for the storage of tuples. The current implementation of RDBSpace uses the MM mySQL JDBC driver and the mySQL relational database management system.

8.4 Built-in SQL support

A 3-Tier model is frequently used in today's e-commerce and enterprise solutions, where normally the first tier is the user interface, the second is the business logic, and the third is the database. Since Tuplespace systems provide high scalability, concurrency, and ease of dynamic networking through loose coupling, Tuplespace systems can actually make an efficient middle tier. These can easily expand to accept new connections and shrink as disconnections occur, without affecting the whole enterprise system. However, one drawback is that Tuplespace systems are not database-centric. Most of the second tier programs focus on generating and executing SQL database queries, but this Tuplespace system's APIs are built with no SQL compatibility. For programmers who are already familiar with database interactions, Tuplespace APIs may appear irrelevant. A further research topic could be to investigate the database issues, possibly incorporating some database operations or SQL support into the Tuplespace API.

As the main idea of the research is to investigate the idea behind the Tuplespace model for distributed computing, the Tuplespace system developed for the research is far away from being a mature system. While there is still much work needed to be done to make the system fully distributed and reliable, the presented prototype did provide a test-bed for the in-depth investigation of transactions in such systems.

9. Conclusions

The proposed Tuplespace system is aimed to provide a framework with transactional controls for building distributed programs which can be implemented without specialized hardware or platform restrictions. With Java's platform independence, message passing, associate searching, remote method invocation, and Tuple-space partitioning, we can build a flexible, scalable, sharable, and reliable object storage system. Transactions have been shown to be a useful tool for adding reliability to distributed systems. When the transactional controls are provided, it is much easier to build applications on top of this Tuplespace system because the reliability and concurrency issues are already taken care of. Nested transactions provide a potentially useful extension over single-level transactions by offering a number of extra features. In conclusion, the Tuplespace architecture is powerful with a small set of easy-to-master methods. Comparing to other models for developing distributed applications, it offers simpler design, more robust, expandable and highly scalable results.

References

[1] Jim Farley. *Java Distributed Computing*. O'Reilly Media, Inc, 1995.

[2] Elliotte Rusty Harold. *Java Network Programming, Third Edition*. O'Reilly Media, Inc, 2005.

[3] Prashant Sridharia. *advanced JAVA networking*. Prentice-Hall, Inc, 1997.

[4] David Gelernter. *Generative communication in Linda*. ACM Trans. Programming Languages and Systems, January 1985.

[5] Nicholas Carriero, David Gelernter. *LINDA IN CONTEXT*. Volume 32, Number4, Communications of the ACM April, 1989.

[6] Bill Venners. *Sway with JavaSpaces: A Conversation with Ken Arnold*. September 30, 2002. http://www.artima.com/intv/sway.html

[7] Philip Bishop, Nigel Warren. *Observing JavaSpace-Based Systems*. October 7, 2002. http://www.artima.com/jini/jiniology/obspaceA.html

[8] Geoffrey C. Arnold, Gregory M. Kapfhammer, and Robert Roos. *Implementation and Analysis of a JavaSpace Supported by a Relational Database*. P950 – 955 ISBN: 1-892512-88-2, CSREA Press, 2002.

[9] Castellani, S, Ciancarini, P. and Rossi, D.. *The ShaPE of ShaDE: a Coordination System*. Technical Report UBLCS, Department of Computer Science, University of Bologna, Italy, 1995.

[10] Martin Gaedke, Klaus Turowski. *Web-Based Federation of Business Application Systems for Ecommerce Applications*. In: S. Conrad; W. Hasselbring; G. Saake (Ed.): 2rid Intl. Workshop on Engineering Federated Information Systems (EFIS99), Germany, 1999.

[11] Umesh Bellur, Siddharth Bondre. *xSpace – A Tuple Space for XML & its application in Orchestration of Web services*. P766 – 772 ISBN: 1-59593-108-2, ACM Press, New York, NY, USA, 2006.

[12] Andrea Omicini. *On the semantics of tuple-based coordination models*. P175 – 182 ISBN: 1-58113-086-4, ACM Press, New York, NY, USA, 1999.

[13] Robert Jellinghaus. *Eiffel Linda: An Object-Oriented Linda Dialect*. P70 – 84 ISSN: 0362-1340, ACM Press, New York, NY, USA, 1990.

[14] Eric Freeman and Susanne Hupfer. *Make room for JavaSpaces, Part I - Part VI.* http://www.artima.com/jini/jiniology/js1.html

[15] Bernhard Angerer. *Space-Based Programming*, 19 March, 2003. http://www.onjava.com/pub/a/onjava/2003/03/19/java_spaces.html

[16] *JavaSpaces^{TM} Service Specification, Version 2.2.* Sun Microsystems, Inc.

[17] *IBM. The TSpaces vision.* URL: http://www.almaden.ibm.com/cs/TSpaces/html/Vision.html

[18] *GigaSpaces Technologies Ltd.* Gigaspaces. URL: http://www.gigaspaces.com/index.htm.

[19] Venu Vasudevan, Sean Landis. *Malleable Services.* Vol.11, No.4, International Journal of Software Engineering and Knowledge Engineering, 2001.

[20] Melinda-Carol Ballou. *DEC to display parallel processing application running on network supercomputer at DECworld.* Vol. 7, No. 26, Digital Review, July 9, 1990.

[21] Robert A. Whiteside, Jerrold S. Leichter. *Using Linda for supercomputing on a local area network.* P192-199 ISBN: 0-8186-0882-X, IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[22] Robert Tolksdorf, Elena Paslaru Bontas, Lyndon J. B. Nixon. *A coordination model for the Semantic Web.* P419 – 423 ISBN: 1-59593-108-2, ACM Press, New York, NY, USA, 2006.

[23] *Semantic Web.* W3C Technology and Society domain, http://www.w3.org/2001/sw/.

[24] *Jini^{TM} Entry Specification, version 1.0.* Sun Microsystems, Inc.

[25] Cay S. Horstmann, Gary Cornell. Core JAVA 2, Volume II – Advanced Features, Seventh Edition. Sun Microsystems, Inc, 2005.

[26] Chandrasekhar Boyapati, Robert Lee, Martin Rinard. *Safe Concurrent Programming in Java.* MIT Laboratory for Computer Science, 200 Technology Square, Cambridge MA, 2000.

[27] Ivor Horton. *Beginning Java^{TM} 2 SDK 1.4 Edition.* Wiley Publishing, Inc, 2003.

[28] Rachid Guerraoui. *Nested Transactions: Reviewing the Coherence Contract.* INFORMATION SCIENCES 84, 161-172, Elsevier Science Inc, 1995

[29] Jean Bacon. *Concurrent Systems Operating Systems, Database and Distributed Systems: An integrated Approach, Second Edition.* Addison Wesley Longman Ltd, 1998.

[30] *Java Transaction API (JTA), Version 1.0.1.* Sun Microsystems Inc, 1999.

[31] Brian Goetz. *Understanding JTS -- An introduction to transactions.* http://www-128.ibm.com/developerworks/java/library/j-jtp0305.html

[32] *Jini^{TM} Transaction Specification, Version 2.0.* Sun Microsystems, Inc.

Appendix A: CD – ROM

The enclosed CD-ROM contains the code of the presented system.