IBM TSpaces

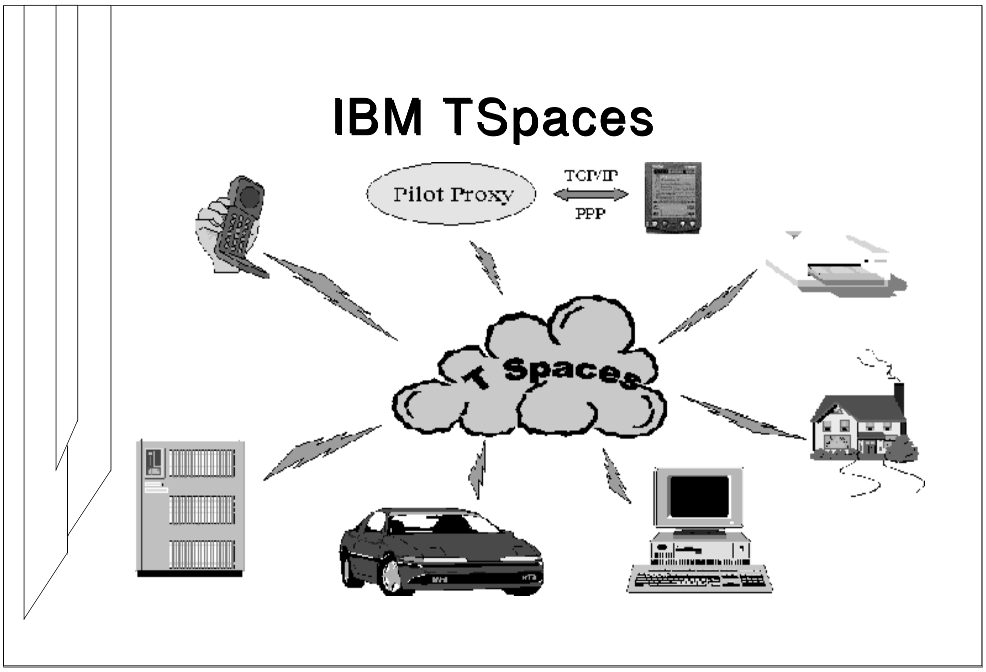
vs

Java Spaces

Bogdan Stanca 26.06.2000

The goal of this presentation is to introduce you to the basics of a TupelSystem.

This slides will present the two wellknown TupleSystems, one is the IBM's TSpaces, and the other one SUN's JavaSpaces.



- Tspaces is a new software that provides a common set of services for a network of heterogeneous computers and operating systems
- SERVICES: group communication, database, URL/based file transfer, event notification
- is tiny (ideal for bringing network services to small and embedded systems)
- By adding additional client applications, it is possible to use TSpaces as an universal print service, email service, pager service, remote control service, and so on.
- Written in Java, TSpace client application can be loaded dynamically
- TSpace server, also written in Java, can be updated while running, avoiding costly downtime for updating

Background

- ☞ In the last 15 years no convenient middleware for sharing data across applications.
- ☞ No network–oriented, easy–to–use data repository
- ☞ Packages were too low level, or too heavy weight e.g. database systems
- ☞ Reason: hard to build on heterogeneous platforms
- ☞ Data should live longer than the program instance that generated it.

- It is hard to build really useful middleware across heterogeneous platforms. One of the hardest problems is just dealing with the representations of objects (e.g., C and C++ objects), which often have differing structure layouts per compiler and per platform. Some attempts have been made to solve this problem alone (e.g., CORBA), but with no satisfying results
- In the past it was a complete successto have two programs start execution, exchange a packet of data, then terminate. The data didn't live any longer than the program instance that generated it. Today's network–oriented application programs require data repositories that can hold the data beyond the life of the generating applications.

Background

- ☞ Java
- ☞ TSpaces
- ☞ TSpaces started out as a marriage of a tuplespace system with a database system

- The solution to the today's needs has several pieces.
- One piece is Java. Java has solved communications and heterogeneosity.
- Another piece is TSpaces. TSpaces together with Java creates a uniform platform on wich we can create applications to perform practically any service desired.
- TSpaces came, actually, from the marriage of a tuplespace system with a database system.

Something Old

- ☞ Mid–80's, David Gelernter (Yale Univ.) created the project Linda
- ☞ Linda is a programming language designed to address the comunication problems of parallel programming.
- ☞ Tuplespace
  - ☞ Anonymous communication
  - ☞ Universal associative addressing
  - ☞ Persistent data

- The LINDA Touplespace was originally intended as a global communication buffer for parallel programs (which are concerned mostly with synchronization and high performance), turned out to work well for distributed programs which often need a persistent communication buffer.
- The central concept of LINDA Touplespace was simple: agents post unstructured tuples to a universally visible Touplespace using an „OUT" command, consume tuples using an „IN" command and read tuples using a „READ" command. LINDA was a hit in the parallel programming comunity, but it has not enjoyed much visibility outside of that particular research area.

Something New

- ☞ In 1996 SUN was showing interest in LINDA, and publicized an internal Tuplespace project, written in Java, called JavaSpaces
- ☞ IBM started some own experiments with Tuplespaces, so ...
- ☞ platform with powerful object oriented types to express any type of data, dynamic class loading(new types), new commands (operators) could be defined to the server. But a server can do more than OUT, IN, and READ.

At the beginning was LINDA ... and at once SUN and IBM started to experiment on that „old“ parallel computing aproach.

Something Borrowed

- ☞ TSpaces thought as being lightweight data repository, so ...
- ☞ Some database features were borrowed from DB2, as data integrity and search speed, transaction support, index support and a simple Query Language.
- ☞ TSpaces server can deal with all the Java types and also with user–defined types, after defining them to the server, ofcourse.

The idea was that they needed something verry fast and rellyable for the persistence and search optiion.

Something Blue

☞ BlueSpaces

☞ TSpaces

- The IBM's favorite choice for the name of this project was Bluespaces (kind of tongue in cheek play on the name JavaSpaces)
- The second choice ended up with TSpaces.

Using TSpaces

- ☞ Network middleware
- ☞ Middleware has no frontend
- ☞ Middleware clients is what the users see
- ☞ Think of TSpaces as
  - ☞ A global persistent communication buffer
  - ☞ A lightweight database
  - ☞ A queue manager (a simple form of IBM's MQ Series)
  - ☞ A dynamic computation engine

- One of the benefits of being attached to a TSpaces server is that you get the services of a real database system(queries, transactions, persistent data, etc.) without the drudgery of dealing with Relational Database System design or SQL queries. To insert data into a TSpace, just drop it in with a write statement (with no schema, no table definition, no ugly JDBC/ODBC statements)

The TSpace Model

```
// Create the initial tuplespace
Tuplespace ts = new Tuplespace(spaceName,
   serverName);
// Write some data, tagged „mydata1"
ts.write(„mydata1", dataInstance);
// Read the specific data record
resultTuple = ts.read(„mydata1", DataInstance);
// Read all the records of that type
resultTupleSet = ts.scan(String, DataInstance);
```

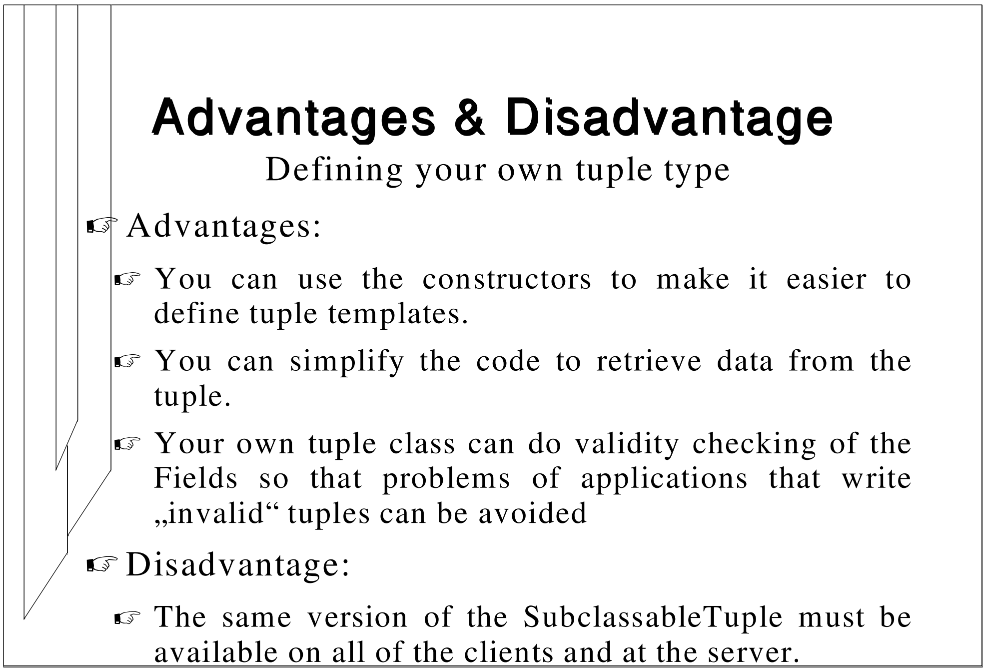
- There are clients and there are servers.
- Clients can read/write data from/to a server
- No table definitions.
- No SQL insert statements
- No decomposing your complex datatypes into SQL primitive types.

- A client can talk simultaneously to several servers, but in the current release, servers do not share information

The Basic Definitions

- ☞ Field
  - ☞ Type
  - ☞ Value
  - ☞ Field Name (optional)
- ☞ Tuple
  - ☞ SuperTuple
  - ☞ Tuple
  - ☞ SubclassableTuple

- A Field object is the most basic component of the Tuplespace data structure hierarchy.
- Specifying a field name implies that this field will be indexed (and searched).
- A Tuple is an ordered sequence of Fields.
- The SuperTuple abstract class contains the functionality of the tuples, so clients should create Tuple or SubclassableTuple objects.
- To define userdefined Tuple types, you can’t extend SuperTuple because this class is final, so you have to extend SubclassableTuple.
- A template tuple is a tuple used for matching. One or more Fields in a template may be „wildcards“ and consist of only the class type with no value.



It is important to note that since a SubclassableTuple is not a subclass of Tuple, one does not have to be concerned that a Tuple template will accidentally match a user defined SubclassableTuple.

The Basic Definitoins

☞ TupleSpace Tuple−Commands

- ☞ write (tuple)
- ☞ take (template_tuple)
- ☞ waitToTake (template_tuple)
- ☞ read (template_tuple)
- ☞ waitToRead (template_tuple)
- ☞ scan (template_tuple)
- ☞ countN (template_tuple)

- write (tuple) Adds a tuple tu space.
- take (template_tuple) Performs an associative search for a tuple that matches the template. When found, the tuple is removed from the space and returned. If none is found, returns null.
- waitToTake (template_tuple) Same as above but blocks until match is found.
- read (template_tuple) Like „take“ but tuple is not removed from the tuplespace.
- waitToRead (template_tuple) Like „waitToTake“ but tuple is not removed from the tuplespace.
- scan (template_tuple) Like „read“, except returns the entire set of tuples that match.
- countN (template_tuple) Like „scan“ but returns only the count of matching tuples.

The 1st Example

```
String host = „tserver1.company.com";
try {
  TupleSpace ts = new TupleSpace(„Ex1", host);
  Tuple t1 = new Tuple(„Key1", „Data1");
  ts.write(t1);
  ts.write( new Tuple(„Key2",new Integer(999),
  „Data2");
  ts.write(„Key2", „Data3");
}catch(TupleSpaceException tse){
  tse.printStackTrace(); }
```

- The above will contact the TSpace server running on the „tserver1" system and connect to the TuplaSpace that is named „Ex1". If the „Ex1" TupleSpace does not exist it will be created at this time.
- After that 3 Tuples will be written to the TupleSpace „Ex1". The 3 Tuples do not really need to all have the same type of Fields.
- The following are all equivalent:

```
Field f1 = new Field(„Key1");
Field f2 = new Field(„Data1");
Tuple t1 = new Tuple();
t1.add(f1);
t1.add(f2);
ts.write(t1);
        ––or––
Tuple t1 = new Tuple(„Key1", „Data1");
ts.write(t1);
        ––or––
ts.write(„Key1", „Data1");
```

The 1st Example

```
try {
   Tuple template = new Tuple („Key2", new
   Field(String.class));
   Tuple tuple = ts.read ( template );
   String data =
   (String)tuple.getField(1).getValue();
}catch(TupleSpaceException tse){
   tse.printStackTrace(); }
```

- Allmost all of the TupleSpace methods can trow TupleSpaceException and therefore one needs to wrap th TSpaces code in Try/Catch blocks.
- Some of the examples that follow, may not show the try/catch in order to make the example easier to follow, but be assured, you MUST have either try/catch blocks or declare that the method trows TupleSpaceException.

Blocking Commands

```
MACHINE 1:
Tuplespace ts = new TupleSpace(„Ex1",host);
SuperTuple answer = ts.waitToTake(„3", new
  Field(String.class));

MACHINE 2:
Tuplespace ts = new TupleSpace(„Ex1",host);
ts.write(„2", „Data2");
Thread.sleep(7500);
ts.write(„3", „Data3");
```

- Assuming that the applications started at the same time, the application on machine 1 would issue the waitToTake and then be blocked. However, 7.5 seconds later as soon as the matching Tuple arrived at the server, the server would send it to machine 1 which would unblock and process the matching Tuple.
- By default blocking commands is to wait forever. However one can specify an optional timeout parameter to specify taht if the operation has not be satisfied when the time limit is exceeded, then the waiting operation should be terminated and the NULL should be returned.
- Here is the modified example that waits only 60 seconds

```
TupleSpace ts = new TupleSpace(„Ex1",host);
Tuple template = new Tuple(„3",
                        new Field(Sting.class));
SuperTuple answer = ts.waitToTake(template,
                                        60*1000);
if (answer == null)
     System.out.println(„Operation timed out.");
```

The 2nd Example

```
UserTuple newTuple;
newTuple = new UserTuple(„Key1“, „Data1“);
ts.write(newTuple);

newTuple = new UserTuple(„Key2“, „Data3“);
ts.write(newTuple);

templateTuple = new UserTuple(„Key2“);
newTuple = (UserTuple)ts.take(templateTuple);
String data = newTuple.getData();
```

First of all two tuples are inserted into the TuplaSpace.
The templateTuple is contructed with one parameter as we shown some slides ago.
The retrieving of the tuple’s data is also simplified by the UserTuple class.
So there are some good pro arguments to user user defined tuples..

The 2nd Example

```
class UserTuple extends SubclassableTuple
implements Serializable {
public UserTuple(String key, String data) {
    super(kay, data); }
public UserTuple(String key) {
    // build template if only one operand
    super(key, new Field(String.class)); }
public String getData() trows
   TupleSpaceException {
    return (String)this.getField(1).getValue();
   }
}
```

The user defined tuple class shows as below.

Implements two constructors, first for new tuples defining, and the second for templates.

The getData function will make retrieving data from the tuple more confortable.

Hello World – Message class – JS

```
import net.jini.core.entry.Entry;
import java.io.Serializable;

public class Message implements Entry {
  public String to;
  public String from;
  public Serializable message;
  public Message() {}

  public Message(String to) {
     this.to = to;
     this.from = null;
     this.message = null; }

  public Message(String to) {
     this.to = to;
     this.from = null;
     this.message = null; }
}
```

JavaSpace implements the Entry Interface

With JavaSpace you define the tuple data that is written to the space with public instance variables.

Hello World – Message class – TS

```
import com.ibm.tspaces;
import java.io.Serializable;

public class Message extends SubclassableTuple {
  public Message() trows TupleSpaceException {
     super(new Field(String.class),
        new Field(String.class),
        new Field(Serializable.class));
  }
  public Message(Sting to) trows
        TupleSpaceException {
     super(to,new Field(String.class),
        new Field(Serializable.class));
  }
  public Message(String to, String from,
        Serializable message)
        trows TupleSpaceException {
     super(to, from , message);
  }
```

TSpaces extends the SubclassableTuple class.

Hello World – Message class – TS

```
// get/set methods for Destination, To and Message
   public String getDestination() throws
        TupleSpaceException {
      return (String)getField(0).getValue();
   }
   public void setDestination(String to) throws
        TupleSpaceException {
      getField(0).setValue(to);
   }

   getFrom() ...
   setFrom(String) ...

   getMessage() ...
   setMessage(String) ...
}
```

With TSpaces accesor methods are used to get and set the tuple Fields.

Allthough the TSpaces implementation is much longer, it is mostly boilerplate get/set methods.

Hello World class – JS

```
import jsbook.util.SpaceAccessor;
import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class HelloWorld {
  public static void main(String[] args) {
    String myName = „World";
    boolean needReply = false;
    if (args.length > 0)
       myName = args[0];
    try {
      JavaSpace s = SpaceAccessor.getSpace();
      Message template = new Message(myName);
      Message msg = (Message)s.takeIfExists(
        template, null, long.MAX_VALUE);
```

Only minor changes required to transform a code written in JavaSpaces to TSpaces..

Hello World class – JS

```
if (msg == null) {
        msg = new Message(„World‟, myName,
           „Hello World‟);
        needReply = true;
      } else
        msg = new Message(msg.from, myName, „Hi‟);
         if (needReply)
            msg = (Message)s.take(template, null,
               long.MAX_VALUE);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

The long.MAX_VALUE is the read/write delay.

Hello World class – TS

```
iport com.ibm.tspaces.*;

public class HelloWorld {

  public static void main(String[] args) {
    String myName = „World";
    boolean needReply = false;

    if (args.length > 0)
       myName = args[0];

    try {
       TupleSpace s = new TupleSpace();
       Message template = new Message(myName);
       Message msg = (Message)s.take(template);
```

The JavaSpace invocation of SpaceAccessor is replaced by a call to the TupleSpace constructor.

The JavaSpace takeIfExists command is replaced by the TSpace take command.

Hello World class – TS

```
            if (msg == null) {
              msg = new Message(„World‟, myName,
                 „Hello World‟);
              needReply = true;
            } else
              msg = new Message(msg.getFrom(), myName,
                 „Hi‟);}
            s.write(msg);
            if (needReply)
              msg = (Message)s.waitToTake(template);
          } catch (Exception e) {
              e.printStackTrace();
          }
      }
  }
```

The JavaSpaces write command is replaced with the TSpaces write command with a slightly different signature.

The JavaSpace read and take with a long wait interval are replaced by TSpaces waitToRead and waitToWrite commands.

Scan – Returning Multiple Tuples

```
Tuple template = new Tuple(„Key2",
    new Field(String.class));
int count = ts.countN(template);
if (count > 0) {
    Tuple tupleSet = ts.scan(template);
    if (tupleSet != null) {
        for (Enumeration e = tupleSet.fields();
            e.hasMoreElements(); ) {
                Field f = (Field)e.nextElement();
                Tuple tuple = (Tuple)f.getValue();
                data =
                  (String)tuple.getField(1).getValue();
.....
            }
        }
}
```

First a templateTuple is created to be used for selecting the Tuples from the server. The next line issues countN to get the count of matches. If one or more matches were found the scan will request the Tuples.

The „tupleSet“ contains one or more fields; the value of each field is one of the tuples returned by the scan. So the next step is to use an Enumeration to extract each of the returned Tuples. The fields() method returns an enumeration that allowed to access each individual Tuple. Once the individual tuple is accessed, the getField(int).getValue() method is used to access the data in the Tuple.

EventRegister

```
Tuple template = new Tuple(„Key2",
   new Field(String.class));

UserCallBack callBack = new UserCallBack();
boolean newTread = true;
int seqNum =
   ts.eventRegister(TupleSpace.WRITE,
   template, callBack, newThread);

.....

ts.eventDeRegister(seqNum);
```

One of the most useful features of TSpaces is the ability to register with the server to be informed when specific types of events occur.

For example, you might want to be informed whenever a Tuple that matches a specific template is written to the TupleServer by any client. With TSpaces this is done by using the TupleSpace.eventRegister() method to indicate the type of enevt you are interested in and specifying the Callback object that you want to have handle the event.

EventRegister

```
Class UserCallBack implements Callback {
 public boolean call(String eventName,
    String tsName, int seqNum, SuperTuple
    tuple, boolean isException) {
  if (!isException) {
    String data =
     (String)tuple.getField(1).getValue();
    .....
  } else {
    ... handle Exception ...
  }
 return false;
 }
}
```

The class that you would like to handle the server callback has to implement Callback and the function „call“.

XML Support

```
TupleSpace ts =
   new TupleSpace(„xmlTest‟, server);

String xml = „<?xml version=‟‟1.0‟‟?>‟+
   +‟<MESSAGE>Hello World!</MESSAGE>‟;

Field xmlF = new XMLField(xml);

Tuple xmlT =
   new Tuple(„myXML Tuple‟, xmlF);

ts.write(xmlT);
.....
```

Motivation:

XML is a flexible metadata language taht can be useful in a variety of contexts. Because of its extensible nature, XML can be used to describe everything from networked devices, corporate command hierarchies, and chemical reactions, to grocery shopping lists and daily calendars. As a text–based language, XML is simple, powerful, and easy to understand. As more and more XML application integrate themselves into business and daily applications, the need to store and query XML documents becomes increasingly evident. While there are very few modular XML storage and query engines in existence, TSpaces is working to provide that functionality in a scalable and fault tolerant fashion.

XML Query

```
.....
String xql = „/MESSAGE‟;
Tuple result = ts.scan(new XMLQuery(xql));
System.out.println(„Result tuples:
 „+result);
.....
```

The supported XML Query Language into TSpaces is a subset of XQL.

But this functionality is not yet complete. Some of the functions who should be implemented are:

- boolean operations connecting multiple conditions (grouping)
- sequences and ordering operators (; and ;;)
- semi–joins
- return operators(? And ??)
- wildcard attributes.

Sumary

- Allows heterogeneous, Java–enabled devices to exchange data with little programming effort
- Data can outlive its producer
- Communication are anonymous
- Communication are asynchronous
- Client data can be loaded dynamicaly

Links

IBM’s TSpaces
http://www.almaden.ibm.com/cs/TSpaces

SUN’s JavaSpaces
http://java.sun.com/products/javaspaces