ANOTHER TAKE ON INTRODUCING JINI AND JAVASPACES

THE SERVICE MODEL AND THE SPACE MODEL

Brian Murphy
Sun Microsystems

Different Distributed Technologies

- Traditional remote procedure call (RPC)
- Common Object Request Broker Architecture (CORBA)
- Java RMI (Java's RPC mechanism)
- Jini
- Enterprise Java Beans (EJB's)
- Servlets
- Web services

Common To All Distributed Systems

- Separate parties communicating across a network
- Mechanism for advertising/finding the system itself
- Mechanism for advertising/finding system services
- Well defined model for interaction
  - Service identification
  - Acquisition and reclamation of resources
  - Communication
  - Coordination of actions
- All embodied in a programming model

Abstractly – Jini Is

- A service-based architecture for building networked systems that can adapt to change
  - Erases the hardware/software distinction
- Examples of change on the network
  - Network entity instances (service/client) – come and go
  - Implementations – new features, bug fixes, etc.
  - Communication protocols
  - Topology
  - Failure – entities or the network itself

Concretely – Jini Is

- A set of specifications
  - Core specifications
  - Standard, non-core specifications (net.jini)
- A programming model
- Contributed implementations
- A community

The Principles Of Jini Technology

- First Axiom: fundamental differences exist between local and remote objects
  - The network cannot be ignored or hidden
- Second Axiom: agreement occurs in the public interface
  - Not the implementation, not the protocol
  - The 'remoteness' of an object is part of the contract
- Fundamental Theorem: agreement occurs in the interface if and only if code is moved

7 Fallacies Of Distributed Computing

Peter Deutsch

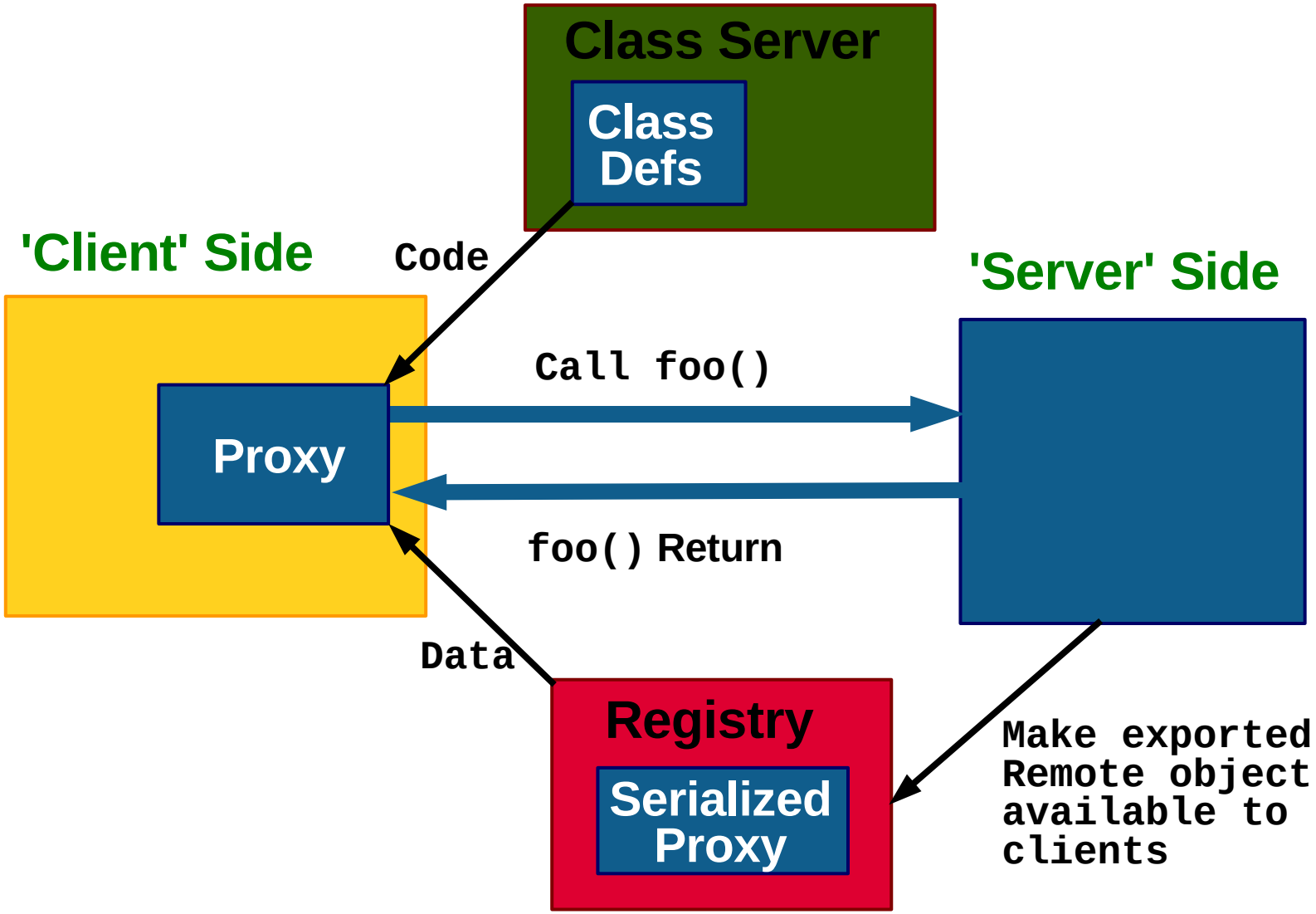
- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Network topology doesn't change
- There is only one administrator
- Transport cost is zero

Communicating Across A Network

The Java Remote Call Model

- Remote object is exported on the server side
  - Produces a proxy to the remote object
- Client side obtains the proxy somehow (RMI Registry, Jini Lookup Service, UDDI, etc.)
  - Code may be downloaded in the process
- Execution of the call is initiated on the client side
- Communication between client and server occurs
- Execution of the call occurs on the server side

The Java Remote Call Model In Action

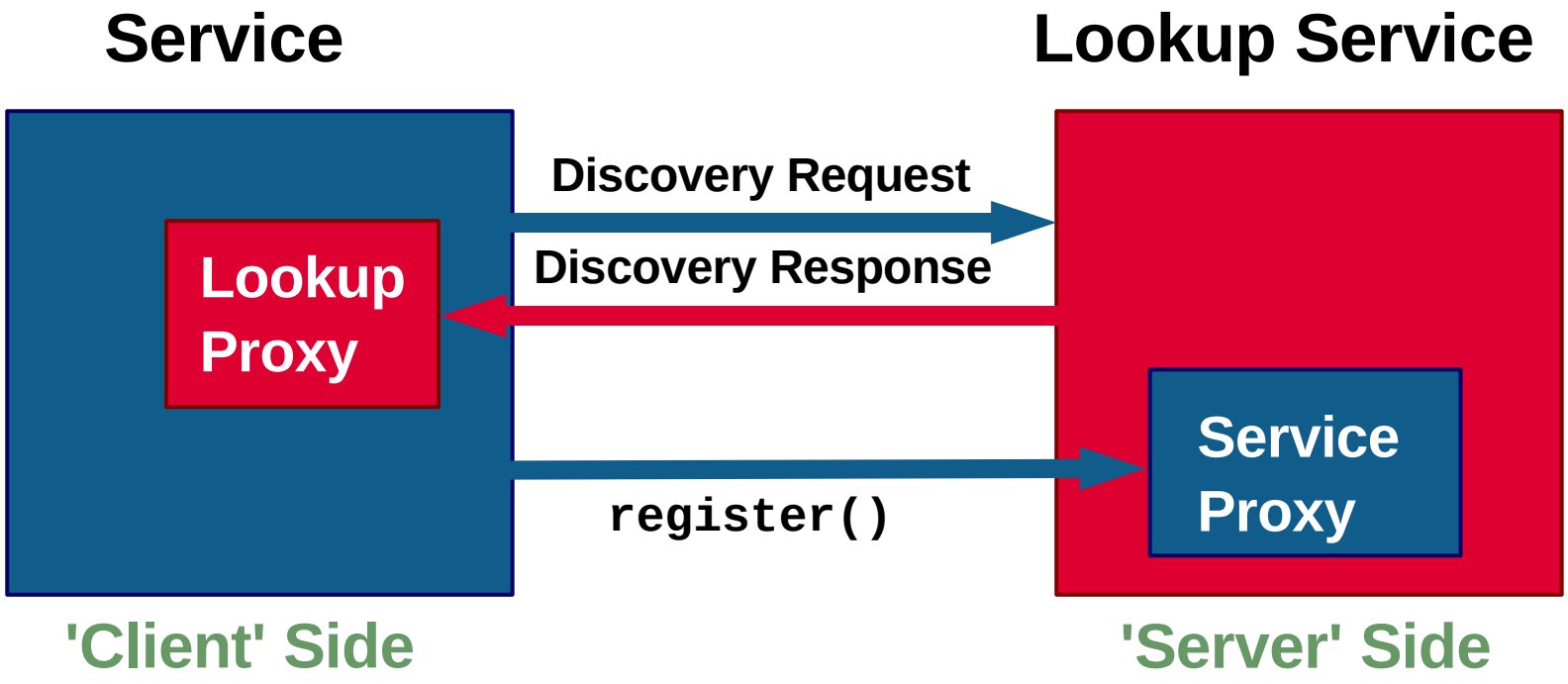


The Jini Programming Model

- Discovery
- Lookup
- Distributed leasing
- Remote events
- Transactions (Two Phase Commit)
- Distributed shared memory (JavaSpaces™) – asynchronous communication
- Comprehensive network security
  - > ***Remote calls and downloaded code***

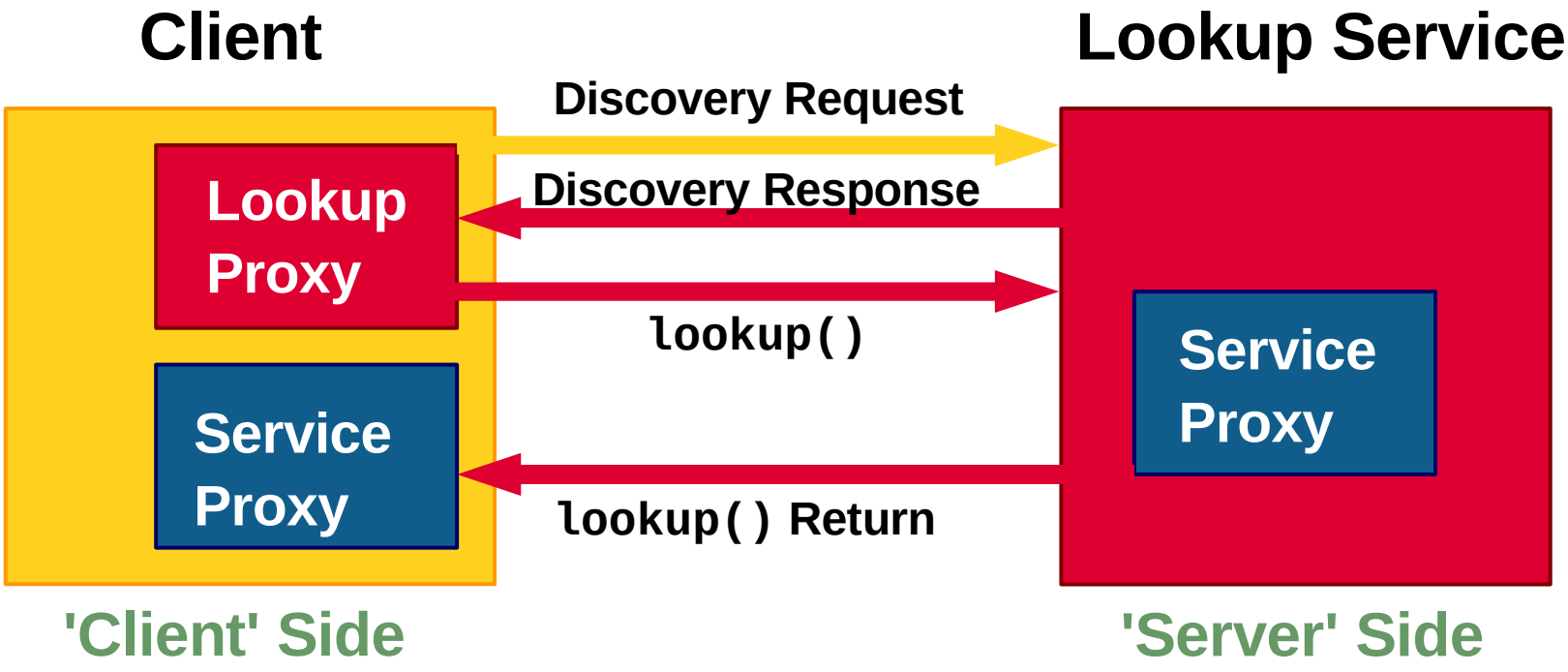
Lookup Discovery And Join

Service and Lookup Service Interaction



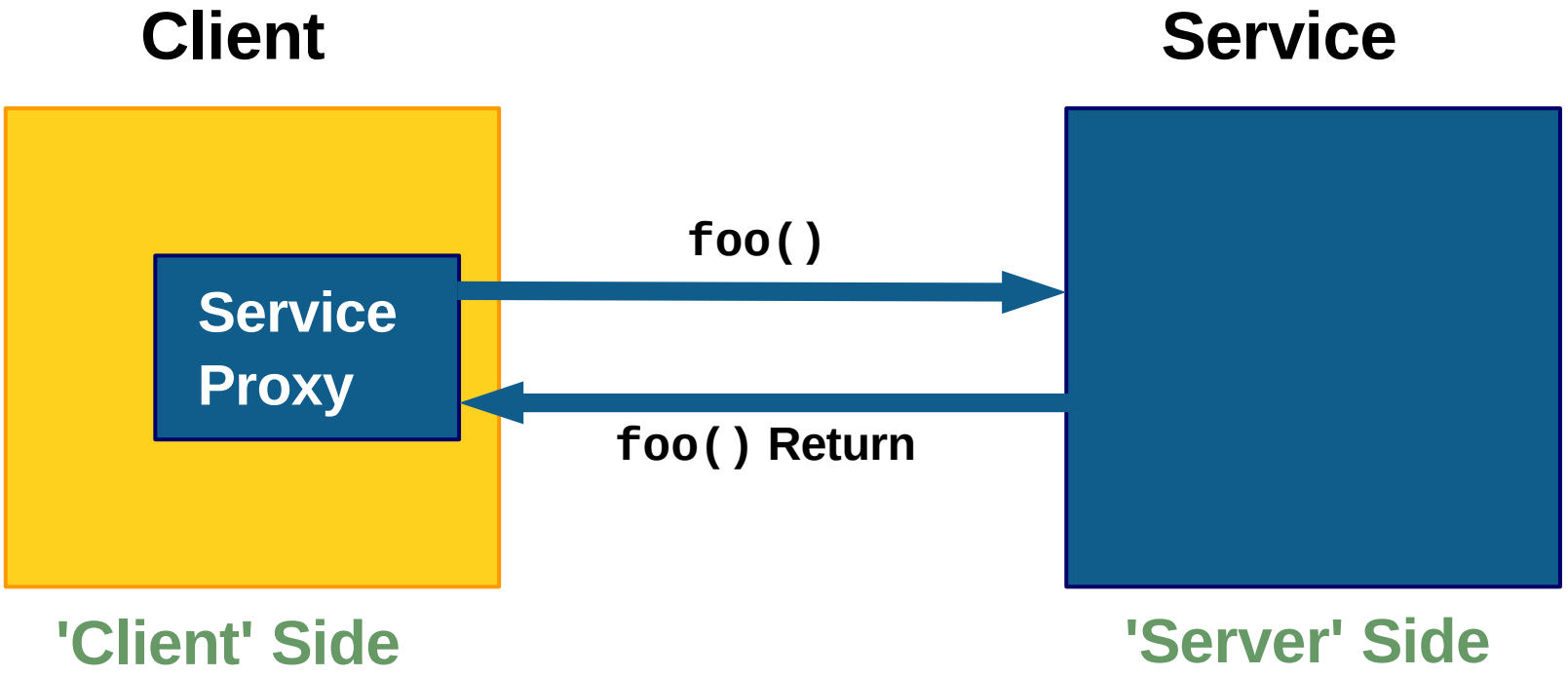
Service Discovery

Client and Lookup Service Interaction



Client And Service Interaction

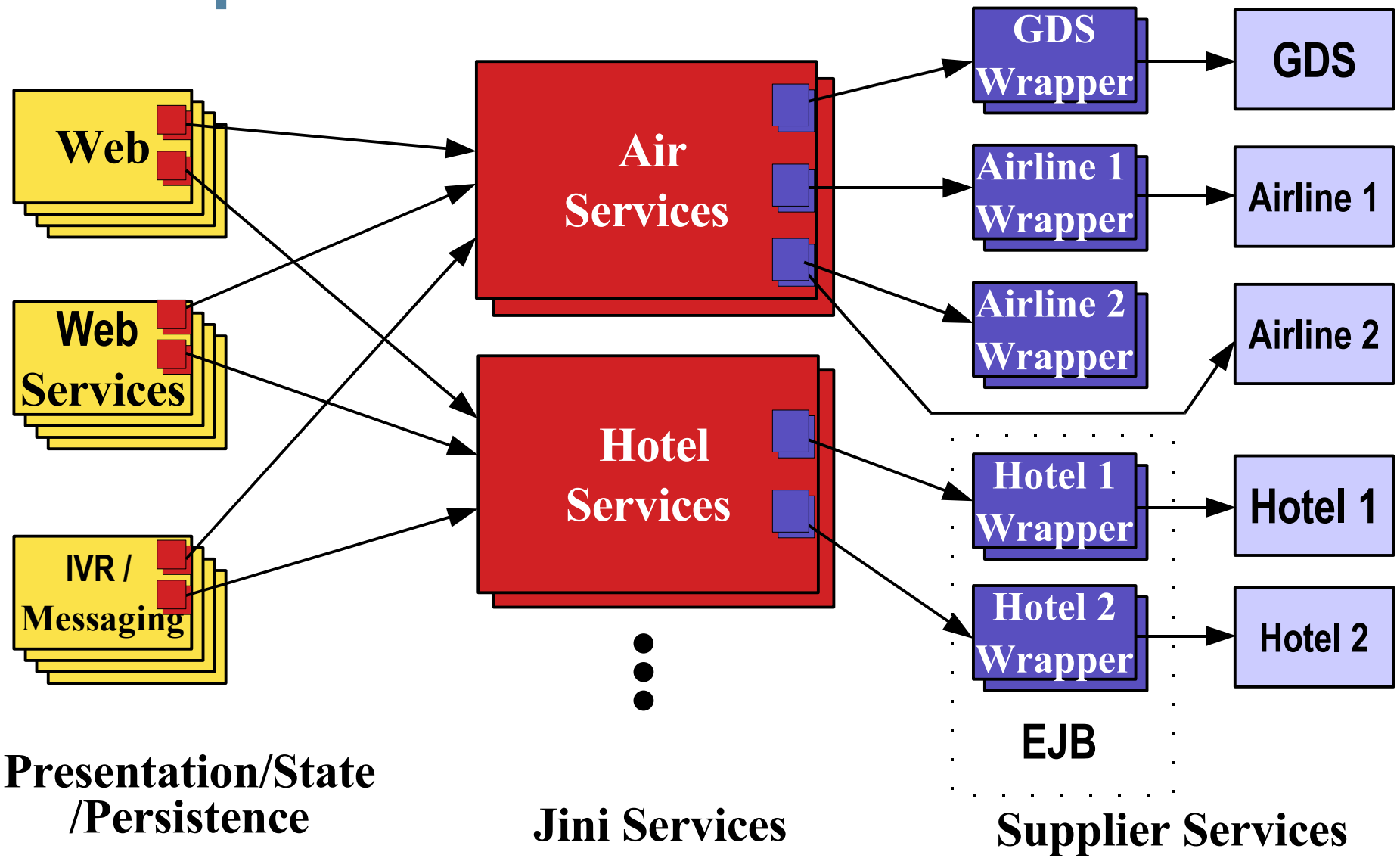
- Code downloaded into the client side
  - Client only knows about service's public interface
- Execution is initiated on the client side
- Execution occurs on the server side



The Jini Service Model

- Clients and services agree on service interface only
- Network protocol is an implementation detail
  - Server side controls both ends of the wire
  - Allows for change over time
    - Bug fixes, requirement changes, new/better implementations
- Exploits code downloading (“smart” proxies)
- Allows for self-healing/self-managing systems
  - Through discovery and leasing
- Security in the face of remote calls/downloaded code

Example: Orbitz.com



An Additional Communication Model

- Sometimes a more decoupled communication model is useful/required
  - Further decouple client and service for less synchronicity
    - Compute server
    - Work flow
    - Message routing
    - Caching/distributed shared memory
- Move the agreement from the service interface to a new, general-purpose interface
- JavaSpaces technology provides this other model

JAVASPACES

A JavaSpace Is A Jini Service

The "*Killer*" Jini Service

- Component of the Jini programming model
- Leverages all of the benefits of the model
  - > Registers with lookup services
  - > Exploits leasing, events, transactions
  - > No wire protocol specified
  - > Exploits code downloading

A Different Communication Model

- Traditional RPC model
  - > Point-to-point – like a telephone
  - > Parties need to know 'who', 'where', 'when'
- The JavaSpaces communication model
  - > Not point-to-point – like a bulletin board
  - > Parties don't necessarily need to know 'who'
  - > Parties don't need to know 'when' – time capsule

Moving The Agreement: Entry Objects

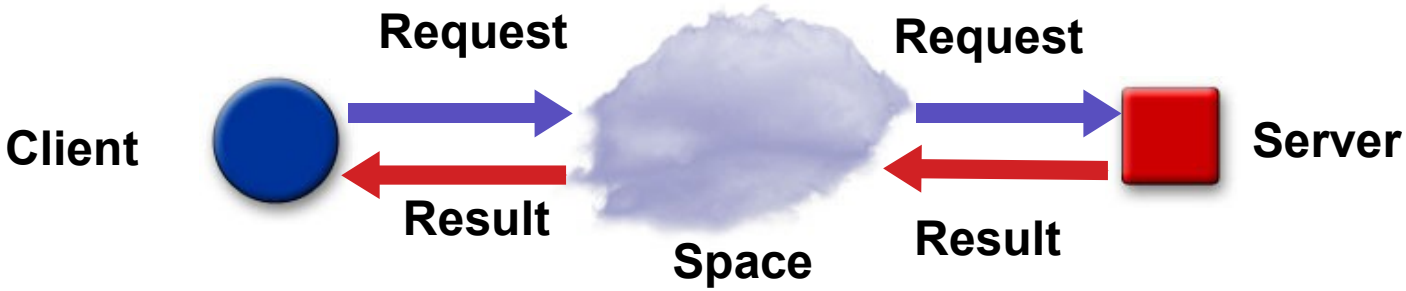
- Jini clients and services agree on service interfaces
- Parties communicating through a “space” agree on what goes in the space
  - Typed collection of objects
    - marker interface – **`net.jini.core.entry.Entry`**
    - public, non-transient, non-static, non-final, non-primitive fields
    - public, no-arg constructor
- Because objects are stored, provides all the benefits of working with objects
  - Real polymorphic value objects
    - Where code movement really “shines”

Simple Yet Powerful API

- Four basic operations
  - **write**
    - Places copy of a given entry into the space
    - Returns a lease on how long entry will be maintained in space
  - **read/readIfExists**
    - Returns copy of an entry from the space that matches a given template (matching semantics similar to lookup service)
  - **take/takeIfExists**
    - Like read, except it removes/returns the entry from the space
  - **notify**
    - Event registration – registers interest in futures entries that match a given template
- Each operation can be executed under a transaction

Example: Compute Server & Spaces

- Perform large, time-consuming computations
  - Break work into small chunks
  - Write an entry for each chunk into the space
  - Wait for results to come back
- Server process loops
  - Taking compute requests
  - Executing each request, writing back results



ComputeTask

Client Writes To Space, Server Takes From Space

```
public class ComputeTask implements Entry {
    public String clientID;
    public InitialConditions data;
    public ComputeTask() { }//no-arg const
    public Entry execute() {
        ComputeTaskResult result =
                        new ComputeTaskResult();
        result.clientID = clientID;
        result.value = data.doComputation();
        return result;
    }
    public long resultLeaseTime() {...}
}
```

ComputeTaskResult

Server Writes To Space, Client Takes From Space

```
public class ComputeTaskResult
                                implements Entry
{
    public String clientID;
    public Computation value;
    public ComputeTaskResult() { }//no-arg const
}
```

Space Based Compute Server Model

- Client writes **`ComputeTask`**s to the space
- Client blocks looking for **`ComputeTaskResult`**s
- Server loops – blocks looking for **`ComputeTask`**s
  - Takes **`ComputeTask`** from the space
  - Executes the task's execute() method
  - Writes **`ComputeTaskResult`** back to the space
- Client takes **`ComputeTaskResult`**s
  - Saves, aggregates, displays, etc. the results

The Client

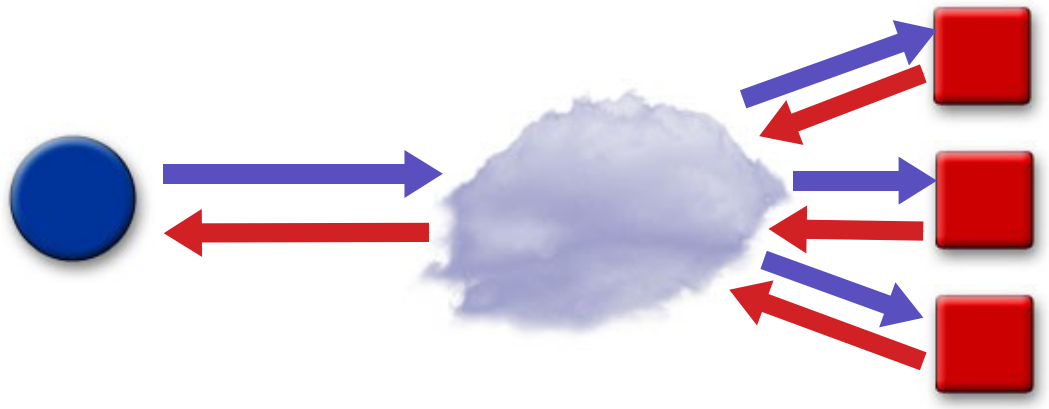
```
int count = 0;
while (i.hasNext()) {//write all tasks
    ComputeTask task = new ComputeTask();
    task.clientID = "seti";
    task.data = (InitialConditions)i.next();
    Lease l = space.write(task, null, 3600000)
    lrm.renewFor(l, Lease.FOREVER, null);
    count++;
}//end loop

ComputeTaskResult template = new ComputeTaskResult();
template.clientID = "seti";
for (int i=0; i<count; i++) {//retrieve/save results
    ComputeTaskResult result =
        (ComputeTaskResult)space.take(template,null,
                                      Long.MAX_VALUE);
    handleResult(result.value);
}//end loop
```

The Server

```
ComputeTask template = new ComputeTask();
while (true) {
     ComputeTask task =
     (ComputeTask)space.take(template,null,
                                  Long.MAX_VALUE);
     Entry result = task.execute();
     if (result != null) {
          space.write(result, null,
                         task.resultLeaseTime());
     }//endif
}//end loop
```

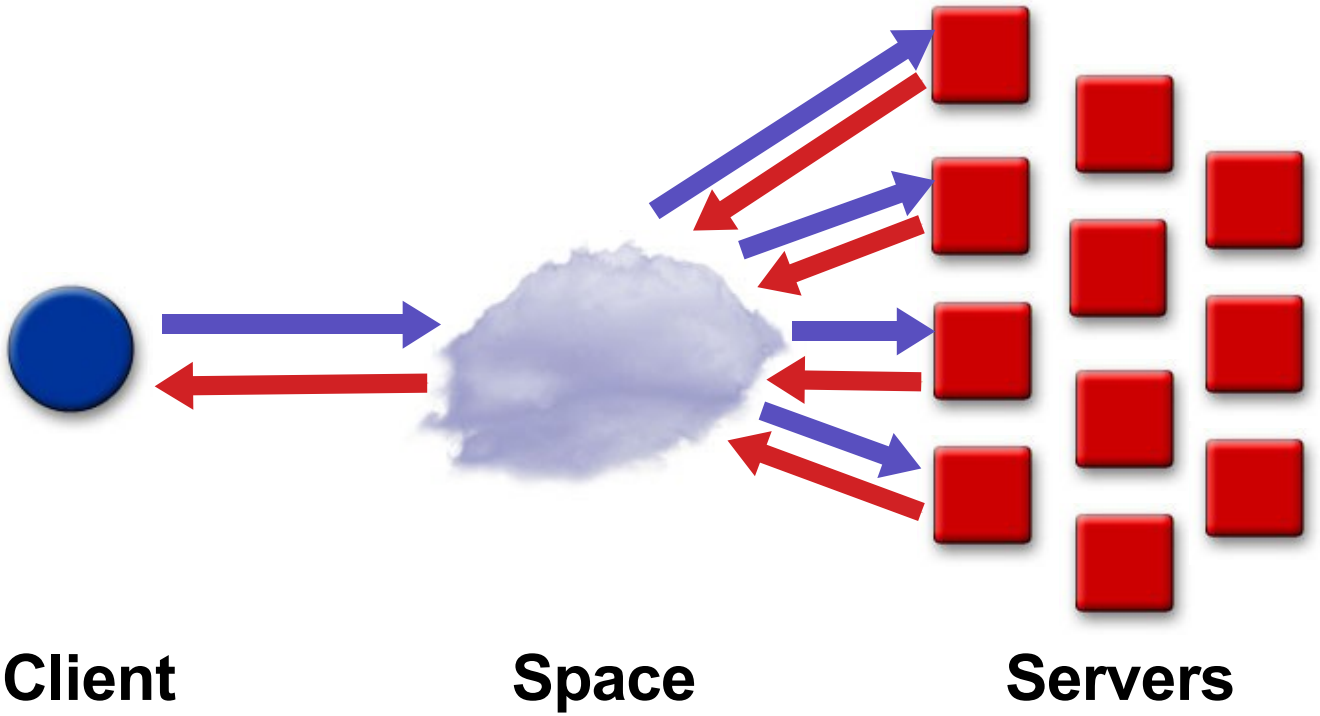
Can Scale Up

- Add more servers...
  - > Don't need to tell client about new servers



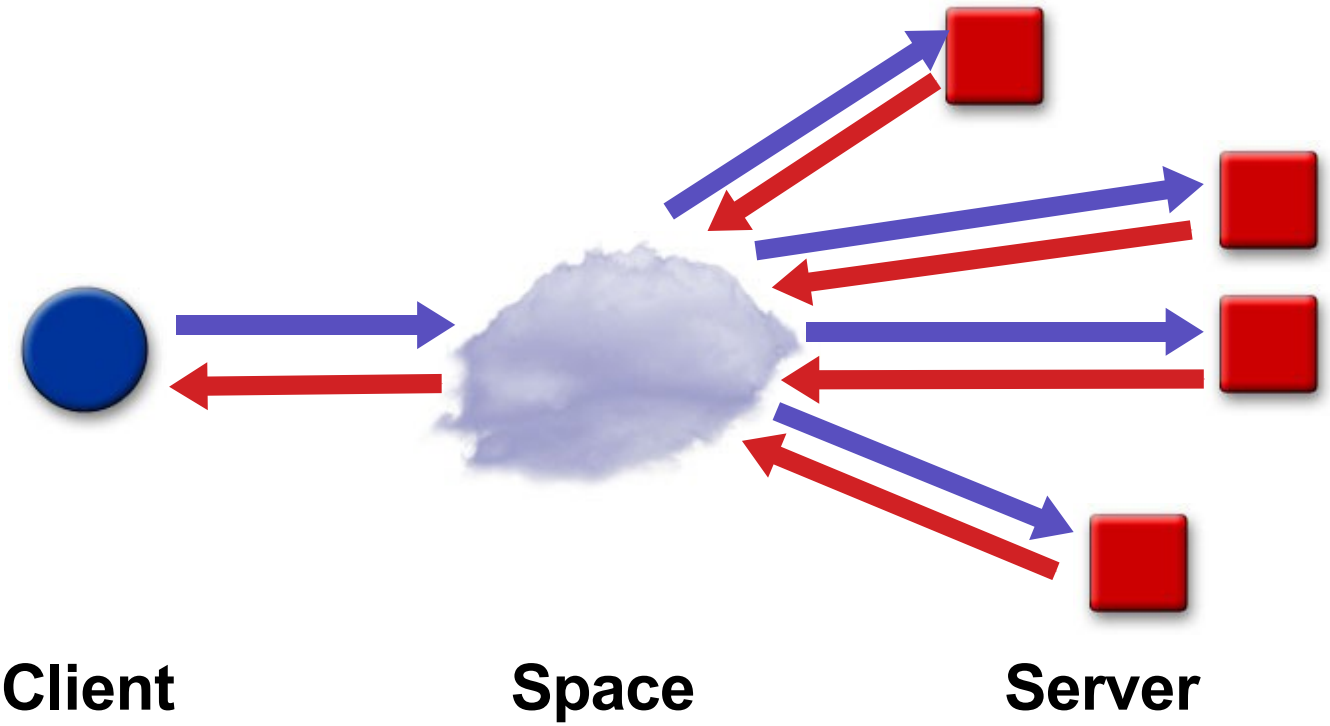
And Up

- Add even more servers



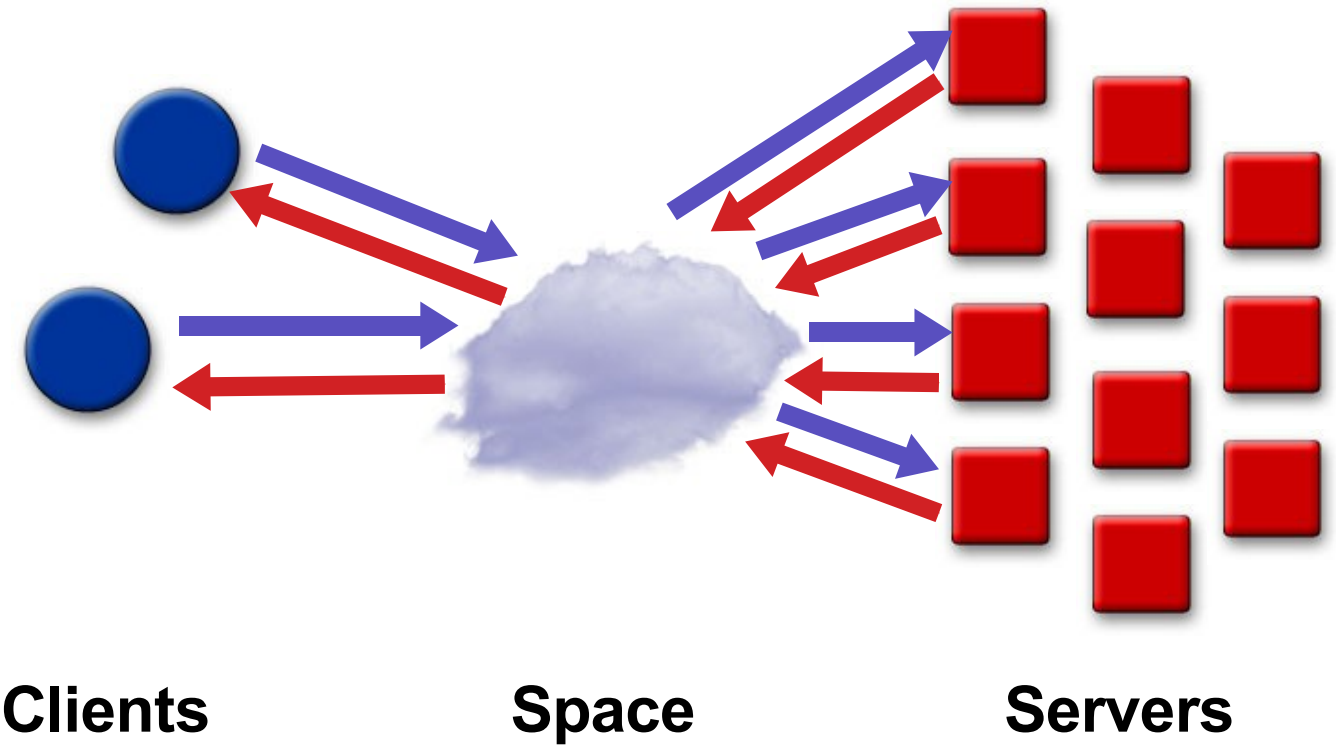
Can Tolerate Failure

- Sometimes servers crash
  - No need to tell client about missing servers
  - Handle planned outages too



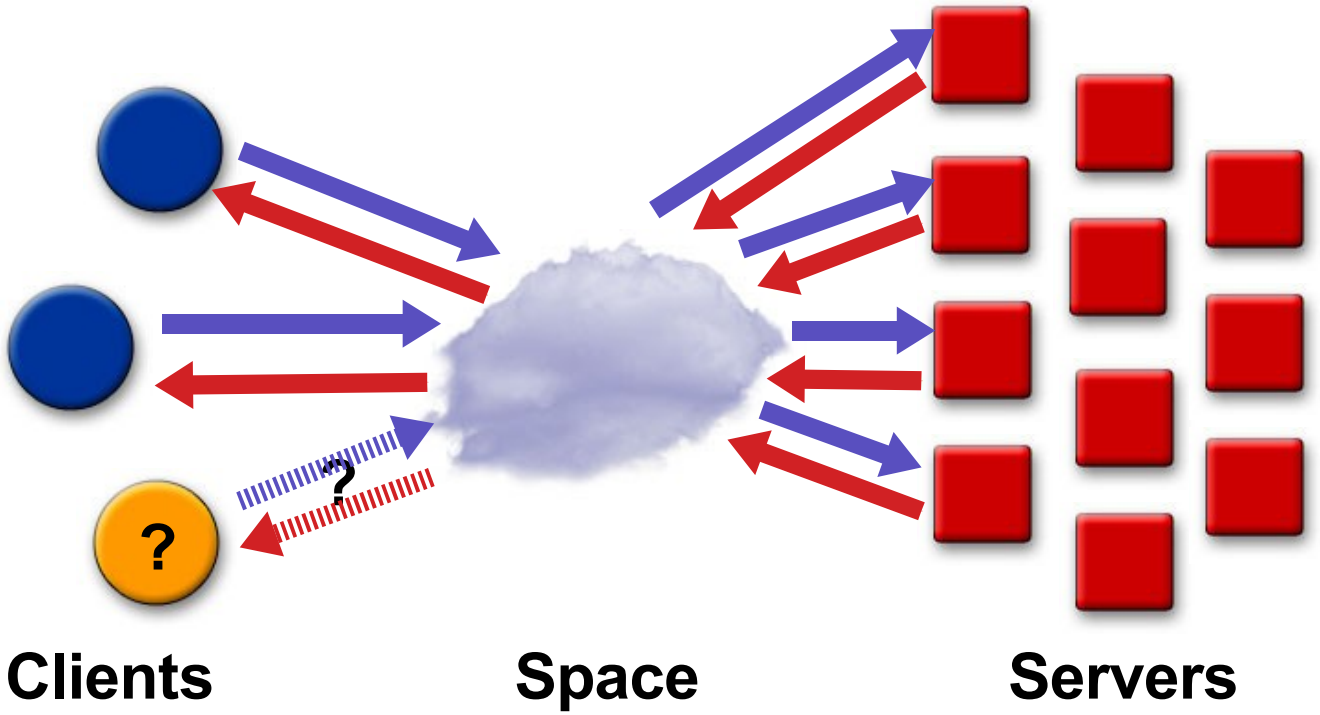
More On Scalability

- Add more clients
  - No need to tell servers about new clients



Adding New Types Of Tasks

- As servers scale up, excess capacity can result
- New kinds of clients with new tasks can be added to exploit the extra available cycles
  - New types of jobs can be run without touching the servers



Agree On A Generic Task Entry

- Server doesn't need to know about **ComputeTask**
- The power of polymorphism
  - > Different subclasses of **Task** class can be written/taken

```
public class Task implements Entry
    public String clientID;
    public Task() {}
    public Entry execute() {
        throw new
          UnsupportedOperationException();
    }
    public long resultLeaseTime() {...}
}
```

Define A New Task To Be Executed

```
public class ComputeTask2 extends Task {

    public Parameters data;
    public ComputeTask2() {}

    public Entry execute() {
        ComputeTask2Result result =
                    new ComputeTask2Result();
        result.clientID = clientID;
        result.answer = data.crunch();
        return result;
    }
    public long resultLeaseTime() {...}
}
```

Small Change To ComputeTask

Extend Task And Remove clientID field

```
public class ComputeTask extends Task {
    // public String clientID;
    public InitialConditions data;
    public ComputeTask() { }//no-arg const
    public Entry execute() {
        ComputeTaskResult result =
                    new ComputeTaskResult();
        result.clientID = clientID;
        result.value = data.doComputation();
        return result;
    }
    public long resultLeaseTime() {...}
}
```

A Few Minor Changes To The Server

Replace ComputeTask With Task

```
Task template = new Task();
while (true) {
    Task task =
            (Task)space.take(template,null,
                                Long.MAX_VALUE);
    Entry result = task.execute();
    if (result != null) {
        space.write(result, null,
                        task.resultLeaseTime());
    }//endif
}//end loop
```

Polymorphism & Code Downloading

- Generic **`Task`** entries not written to the space, only
  - **`ComputeTask`**
  - **`ComputeTask2`**
- The power of polymorhism
  - Clients, server, and space agree on **Entry**
  - Clients and server agree on **Task**
  - Only the clients know about their specific task type
- The power of code downloading
  - The implementation of **`execute`** in **`Task`** never runs
  - Through code downloading, client provides server with the correct implementation of **execute** method

Uses For The Space Model

- Coordinate loosely coupled collections of processes
  - Allows components to come and go
  - Allows components to change
- When problem can be modeled as flow of objects
  - Workflow
  - Dataflow
- Large, parallelizable computations
- Don't use a “space” when you need
  - A file system or a database (object or relational)

Summary

- Jini starts from fundamental principles
  - The network cannot be ignored
  - Agreement is through the interface
  - Code is moved
- Provides a programming model that supports
  - The service model
  - The space model
  - Network security
- Through JavaSpaces, supports the space model
  - Decouple communication through the space
  - Minimize the amount of prior knowledge necessary
    - Exploits polymorphism and code downloading

ANOTHER TAKE ON INTRODUCING JINI AND JAVASPACES

THE SERVICE MODEL AND THE SPACE MODEL

Brian Murphy
brian.t.murphy@sun.com