8 Coordination-Based Systems

8.1 Coordination Models

Coordination models

Essence

We are trying to separate computation from coordination; coordination deals with all aspects of communication between processes, as well as their cooperation.

Couplings

Make a distinction between

- Temporal coupling: Are cooperating/communicating processes alive at the same time?
- Referential coupling: Do cooperating/communicating processes know each other explicitly?

Coordination models

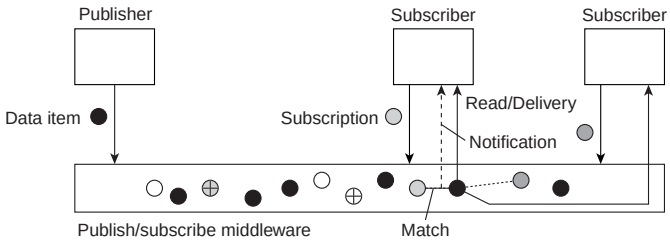
Referential \ Temporal	Coupled	Decoupled
Coupled	Direct	Mailbox
Decoupled	Meeting oriented	Generative communication

8.2 Architectures

Architectures: Overview

Essence

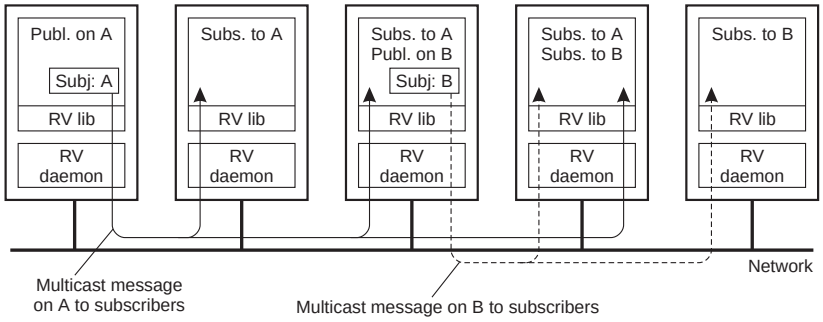
- A data item is described by means of attributes.
- When made available, it is said to be published.
- A process interested in reading an item, must provide a subscription: a description of the items it wants.
- Middleware must match published items and subscriptions.



Example: TIB/Rendezvous

Coordination model
Uses of subject-based addressing ⇒ publish-subscribe system.

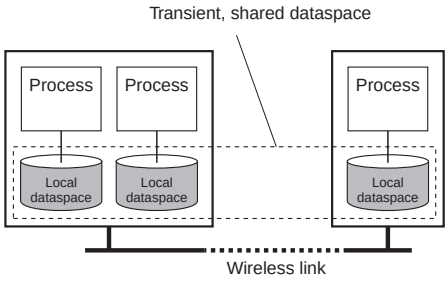
- Receiving a message on subject X is possible only if the receiver had subscribed to X
- Publishing a message on subject X ⇒ message is sent to all (currently running) subscribers to X.



Example: Lime

Lime
Every node has its own dataspace:

- When P and Q are in each other's proximity, dataspaces become shared
- Published data items are stored locally, until removed
- P can publish data items from specific process
- Reactions describe what to do when a match is found



8.3 Communication

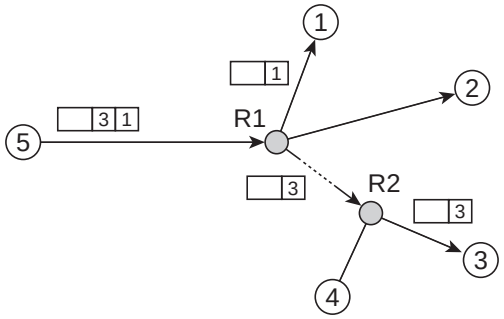
Content-based routing

Observation
When a coordination-based system is built across a wide-area network, we need an efficient routing mechanism (centralized solutions won't do).

Solution

- Naive: Broadcast subscriptions to all nodes in the system and let servers prepend destination address when data item is published
- Refinement: Forward subscriptions to all routers and let them compute and install filters.

Content-based routing: naive solution



8.4 Jini

Jini: Overview

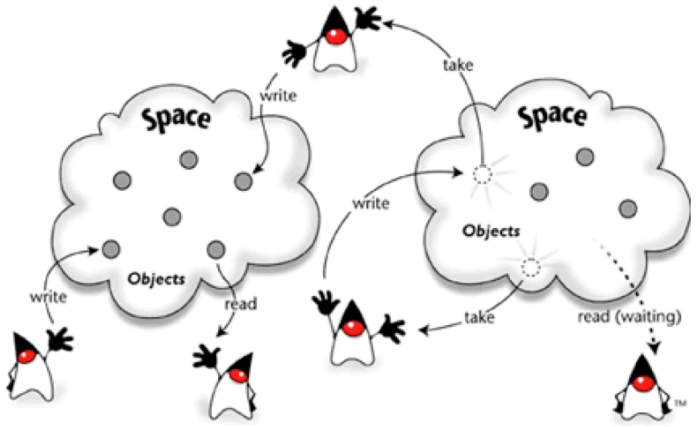
Coordination-based system from Sun Microsystems

- Written in Java: one language everywhere
- Uses RMI and Java Object Serialization to enable Java objects to move around the network
- Offers network plug and play of services (Java objects)
- Services may come and go without administration and reconfiguration
- Federation, not central control
- Programming interfaces designed for robustness

Jini: Main Components

- Service: an entity that another program, service or user can use. It can be a piece of computation, a hardware device or software.
- Client: a Jini device or component that becomes a member of the federation in order to use a Jini service.
- Lookup Service: keeps track of the services offered in the federation.
  - Repository of available services.
  - Stores each service as Java objects.
  - Clients download services on demand.

Jini: Javaspaces



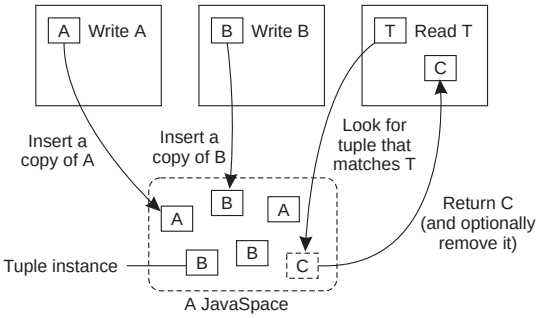
Clients and services have to exchange information and coordinate their activity.

Jini: Javaspaces

Coordination model
Temporal and referential uncoupling by means of JavaSpaces, a tuple-based storage system.

- A tuple is a typed set of references to objects
- Tuples are stored in serialized, that is, marshaled form into a JavaSpace
- To read a tuple, construct a template, with some fields left open
- Match a template against a tuple through a field-by-field comparison

Jini: Javaspaces



Write: A copy of a tuple (tuple instance) is stored in a JavaSpace

Read: A template is compared to tuple instances; the first match returns a tuple instance

Take: A template is compared to tuple instances; the first match returns a tuple instance and removes the matching instance from the JavaSpace

Jini: Terminology

- Spontaneous networking: Communication is established dynamically without installing drivers and carrying out manual configuration
- Federation: A set of software components and devices creating a distributed system that are part of a Jini network at a given time.
- Discovery: The mechanism used to locate lookup services in order to advertise a new service in the network or find a service for use.
- Leasing: Jini services grant resource usage in a time-based manner. If the period of the grant (lease) is not renewed before its expiration, the grant will be withdrawn at the end of the period.
- Distributed event: Components of a Jini system can notify each other when some change in their state occurs.
- Group: Names used to represent a community

Jini: Behaviour

The fundamental behaviour is defined by three protocols:

- Discovery: how to locate the lookup service
- Join: how to register with the lookup service and export services
- Lookup: how to find suitable services

Jini: Operation

- Services export their services (in the form of Java objects)
- Clients locate services and download objects for execution
- Client-Service interaction (formation of a federation) is governed by need
- Lookup services are dynamically discovered by clients and services
- Services register service proxies in Jini lookup services
- Clients lookup and download service proxies from discovered Jini lookup services by interface and/or attributes

Jini: Discovery

Enables clients and services to locate lookup services.

- Client discovery
  - At startup; problem with latecomer services
- Service announcement

  - At startup; problem with latecomer clients

Two forms of discovery:

- Multicast: using UDP multicast
  - Finding services at unknown but multicast-reach locations using group names
- Unicast: using TCP/IP
  - Finding services at known locations
  - URL: jini://hostname:port/

Proxy object of lookup service gets loaded to discovering entity.

Jini: Join

- Service provider already received a proxy of the lookup service
- Provider uses this proxy to register its service
- Gives the lookup service:
  - its service proxy
  - attributes that further describe the service
- Provider can now be found and used in this Jini federation

Jini: Lookup

- Client already received a proxy of the lookup service
- Client uses this proxy to look for a service
- Client creates template: describes the type of service sought after
- Client sends template to lookup service
- Lookup service performs template matching and returns result
  - Strict matching: using marshalled objects for comparing fields
  - Conditional matching:
    * identical service identifiers
    * service is instance of template
    * service attributes contain at least one match for each attribute in template

Jini: Leases

- Time-based grants of resources or services.
- Provides a method of managing resources in an environment where network failures can, and do, occur
- Loose contracts between granter and holder.
- Negotiated for a set period of time.
- Can be shared or exclusive.

Jini: Distributed Events

- Enables Java event model to work in a distributed network.
- Register interest, receive notification.
- Allows for use of event managers.
- Can use numerous distributed delivery models (push, pull, filter...).
- Uses leasing protocol.

Jini Example: Hello World Interface

```
// This is the interface that the services proxy implements

public interface HelloWorldServiceInterface
{
    public String getMessage();
}
```

Jini Example: Hello World Server

```
// The HelloWorld service that returns a string when asked by clients.

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import java.util.Hashtable;
import java.io.IOException;
import java.io.Serializable;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

class HelloWorldServiceProxy implements Serializable,
    HelloWorldServiceInterface
{
    public HelloWorldServiceProxy() {}

    public String getMessage() {
        return "Hello, world!";
    }
}
```

Jini Example: Hello World Publisher

```
// HelloWorldService is the "wrapper" class that handles publishing the
// service item.

public class HelloWorldService implements Runnable
{
    // 10 minute leases

    protected final int LEASE_TIME = 10 * 60 * 1000;
    protected Hashtable registrations = new Hashtable( ;
    protected ServiceItem item;
    protected LookupDiscovery disco;

    // Inner class to listen for discovery events

    class Listener implements DiscoveryListener
    {
        // Called when we find a new lookup service.

        public void discovered(DiscoveryEvent ev)
        {
            System.out.println("discovered a lookup service!");
            ServiceRegistrar[] newregs = ev.getRegistrars();
            for (int i=0 ; i<newregs.length ; i++) {
                if (!registrations.containsKey(newregs[i])) {
                    registerWithLookup(newregs[i]);
                }
            }
        }
```

Jini Example: Hello World Publisher

```
        // Called ONLY when we explicitly discard a lookup service, not
        // "automatically" when a lookup service goes down

        public void discarded(DiscoveryEvent ev)
        {
            ServiceRegistrar[] deadregs = ev.getRegistrars();
            for (int i=0 ; i<deadregs.length ; i++) {
                registrations.remove(deadregs[i]);
            }
        }
    }
```

Jini Example: Hello World Publisher

```
    public HelloWorldService() throws IOException {
        item = new ServiceItem(null, createProxy(), null);

        // Set a security manager

        if (System.getSecurityManager() == null) {
            System.setSecurityManager (new RMISecurityManager());
        }

        // Search for the "public" group.

        disco = new LookupDiscovery(new String[] { "" });

        // Install a listener.

        disco.addDiscoveryListener(new Listener());
    }
```

```
    protected HelloWorldServiceInterface createProxy() {
        return new HelloWorldServiceProxy();
    }
```

Jini Example: Hello World Publisher

```
    protected synchronized void
        registerWithLookup(ServiceRegistrar registrar) {
            ServiceRegistration registration = null;

            try {
                registration = registrar.register(item, LEASE_TIME);
            } catch (RemoteException ex) {
                System.out.println("Couldn't register: " + ex.getMessage());
                return;
            }

        if (item.serviceID == null) {
            item.serviceID = registration.getServiceID();
            System.out.println("Set serviceID to " + item.serviceID);
        }

        registrations.put(registrar, registration);
    }
```

Jini Example: Hello World Publisher

```
    // This thread does nothing but sleep, but it makes sure the VM
    // doesn't exit.

    public void run() {
        while (true) {
            try {
                Thread.sleep(1000000);
                 } catch (InterruptedException ex) {
            }
        }
    }

    // Create a new HelloWorldService and start its thread.

    public static void main(String args[]) {
        try {
            HelloWorldService hws = new HelloWorldService();
            new Thread(hws).start();
        } catch (IOException ex) {
            System.out.println("Couldn't create service: " +
                                        ex.getMessage());
        }
    }
}
```

Jini Example: Hello World Client

```
// A simple Client to exercise the HelloWorldService

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import java.util.Vector;
import java.io.IOException;
import java.rmi.RemoteException;
```

```
import java.rmi.RMISecurityManager;

public class HelloWorldClient implements Runnable {
    protected ServiceTemplate template;
    protected LookupDiscovery disco;

    // An inner class to implement DiscoveryListener

    class Listener implements DiscoveryListener {
        public void discovered(DiscoveryEvent ev) {
            ServiceRegistrar[] newregs = ev.getRegistrars();
            for (int i=0 ; i<newregs.length ; i++) {
                lookForService(newregs[i]);
            }
        }
        public void discarded(DiscoveryEvent ev) {}
    }
```

Jini Example: Hello World Client

```
    public HelloWorldClient() throws IOException {
        Class[] types = { HelloWorldServiceInterface.class };

        template = new ServiceTemplate(null, types, null);

        // Set a security manager

        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        // Only search the public group

        disco = new LookupDiscovery(new String[] { "" });

        // Install a listener

        disco.addDiscoveryListener(new Listener());
    }
```

Jini Example: Hello World Client

```
    // Once we've found a new lookup service, search for proxies that
    // implement HelloWorldServiceInterface

    protected Object lookForService(ServiceRegistrar lusvc) {
        Object o = null;

        try {
            o = lusvc.lookup(template);
        } catch (RemoteException ex) {
            System.err.println("Error doing lookup: " + ex.getMessage());
            return null;
        }

        if (o == null) {
            System.err.println("No matching service.");
            return null;
        }

        System.out.println("Got a matching service.");
        System.out.println("It's message is: " +
                           ((HelloWorldServiceInterface) o).getMessage());
        return o;
    }
```

Jini Example: Hello World Client

```
    // This thread does nothing--it simply keeps the VM from exiting while
    // we do discovery.

    public void run() {
        while (true) {
            try {
                Thread.sleep(1000000);
            } catch (InterruptedException ex) {
            }
        }
    }

    // Create a HelloWorldClient and start its thread

    public static void main(String args[]) {
        try {
            HelloWorldClient hwc = new HelloWorldClient();
            new Thread(hwc).start();
        } catch (IOException ex) {
            System.out.println("Couldn't create client: " +
                               ex.getMessage());
        }
    }
}
```