Flashgridding with Java: Using Project GlassFish℠, JavaSpaces™ and Groovy in an Open Source Supercomputer

Van Simmons, Sean Merritt and Jim Gammill

Project Leaders
ComputeCycles Project
http://computecycles.dev.java.net

TS-3714

Goal of This Talk

Provide information and examples on using the ComputeCycles project, Project GlassFish℠, Jini™ network technology, and Groovy to create a self-assembling supercomputer

Agenda

What Problems Are We Trying to Solve?

What Are the Goals of the ComputeCycles Project?

What are the Components of the ComputeCycles Project?

How Does the ComputeCycles Project Work?

Agenda

What Problems Are We Trying to Solve?

What Are the Goals of the ComputeCycles Project?

What are the Components of the ComputeCycles Project?

How Does the ComputeCycles Project Work?

Grid-Based Computation Is Too Hard

- Too hard to deploy
  - How do you deploy your code to 6K machines you don't control and whose architecture you can't specify?
- Too hard to secure
  - How do you control access to your data and your code in a public, shared environment?
- Too hard to manage
  - How do you stop a runaway process on 6K machines?
- Too hard to debug
  - Whose debugger works on 6K processors at once?
- Too hard to write
  - What parallel programming model do you use in a grid?

Flashgridding

What do we mean by that term?

- Addressing the hard problems means that you can dispense with a lot of grid infrastructure
- We've tried to imagine what a grid with NO permanent infrastructure would look like
  - But we can't get all the way there
- If we can make it easy to deploy, secure, manage, et al., the software in the grid then you can assemble a grid "in a flash"
- The idea behind Flashgridding is to reduce the need for infrastructure to a bare minimum, while using it if it's there

Some Implications of "Flashgridding" When It's Done with Java Technology

- Reduces infrastructure requirements by
  - Allowing heterogeneous hardware
  - Allowing heterogeneous operating systems
- Can provide easy, fast deployment
  - Java technology is everywhere, after all
  - No native code enables write once, run everywhere
  - Eases deployment by loading code from the network
    - Requires creative use of URLClassLoader
- Is able to utilize remote grid resources over the Internet and still be secure
  - Managed code is critical for flashgridding

Why Not Other Open Source or Commercial Grid Products?

Or just WebServices for that matter?

- Most open source projects are, understandably, focused on reusing native code implementations
  - Native code is a huge barrier to flashgridding, because it makes an infrastructure assumption
- Many commercial products are difficult to evaluate or pilot or deploy freely
  - Who signs the license for a commercial product on a machine that just joined your grid across the Net?
  - One vendor required an NDA just to see their API
- Pure web service-based approaches deliberately ignore the possibilities inherent in mobile code

Agenda

What Problems Are We Trying To Solve?

What Are the Goals of the ComputeCycles Project?

What Are Its Components?

How Does the ComputeCycles Project Work?

Peter Deutsch's Canonical 7 Fallacies

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

ComputeCycles Goals

Build a minimal infrastructure compute grid

- Disallow single-point-of-failure components
- Be secure (especially with downloaded code)
- Be largely self-administering
- Be dynamically configured
- Facilitate use of Master/Worker programming model (and eventually workflow)
- Run through firewalls and across the net
- Deploy to broad array of compute platforms
- In short, seek to avoid most of Deutsch's fallacies and address the "too hard" aspects of the grid

Agenda

What Problems Are We Trying To Solve?

What Are the Goals of the ComputeCycles Project?

What Are Its Components?

How Does the ComputeCycles Project Work?

What Sort of Components Are Needed To Meet the Goals?

- Need to provide hosting for processes which
  - Are long-lived
  - Require an auditable security model
  - Can be remotely managed using readily available tools (think web browsers)
- Need to provide a good base for Master/Worker programming model
- Need to provide support for dynamic reconfiguration
- Preferably would be standards-based
- Must be open source

Components of ComputeCycles

We're standing on the shoulders of giants

- The infrastructure elements used to achieve our goals:
  - Project GlassFish (Application and Web Server)
  - Jini network technology (Tuple space implementation for M/W pattern)
  - Groovy (Configuration Language)
  - Java™ technology (Security model, universal deployment platform)
- ComputeCycles is a relatively thin layer of Jini network technology services, Project GlassFish webapps and Groovy config files wrapped around these base components
- N.B. We can't use these components "stock"

Project GlassFish

- Using Project GlassFish for
  - Java Naming and Directory Interface™ API and Lookup Service Discovery
  - Security configuration webapp hosting
  - Service configuration webapp hosting
  - Jini service hosting (services run in Project GlassFish Java VM)
  - Gridapp configuration hosting
  - Hosting EJB™ beans related to service state
- Why Project GlassFish?
  - Primarily for Integration w/NetBeans™ software
- What changes did we need?
  - Incorporate our own URLStreamHandlerFactory

Jini Network Technology

"Jini is all about remote execution and downloadable code", *Brian Murphy, 2003–2006*

- Using Jini network technology for
  - Jini Extensible Remote Invocation (ERI)
  - JavaSpaces™ Technology Kit-provided services (Lookup Service, Transaction Manager)
  - The tuplespace implementation, i.e., JavaSpaces
  - Our own ComputeCycles services
- Why Jini network technology?
  - Secure, downloadable code
- What changes did we need?
  - Dynamic truststore specification

Groovy

"Groovy killed the XML File", *Michael Henderson, 2004*

- Using Groovy for
  - Configuration (in place of XML and Jini based configuration files)
  - To drive our standalone UI when services run outside of an App Server
- Why Groovy?
  - Closure implementation ideal as a substitute for Jini based config components
- What changes did we need?
  - None yet, but may require changes for timing of closure evaluation

Java Technology

- Using Java technology for
  - Universal deployment platform
  - Security model
- Problems we've encountered that are Java SE-related
  - Inappropriate use of system properties
    - javax.net.ssl.trustStore should not necessarily be a singleton
  - Inappropriate lack of use of system properties
    - URLStreamHandlerFactory is a singleton and must be coded rather than supplied dynamically

Agenda

What Problems Are We Trying To Solve?

What Are the Goals of the ComputeCycles Project?

What Are Its Components?

How Does the ComputeCycles Project Work?

Security

- Authentication
- Authorization
- Encryption
- Separation of security domains

Authentication

- "Keyring" based authentication
  - Secret key to gain access to keyring of key pairs
- Currently based on using keytool files
- Keytool files are located at configured URLs
  - Special webapp provided to vend keytool and Java Authentication and Authorization Service (JAAS) files (keystore, truststore, login and password)
- Security configuration is provided to services via a webapp running in a secured environment
  - Https
  - Policy files

Authorization

- Policy based
  - All services run either inside of Project GlassFish or standalone
  - Never run without a SecurityManager in place
- Uses dynamic permission grants like those found in Jini network technology
- Grants are logged and visible to end user
- We've created a special version of the Jini based DebugPolicyProvider to help make this more transparent (and debuggable)

Encryption

- SSL used to access keyring
- SSL used during service execution for all intra-grid communication
- Enables use of geographically spread resources and the internet

Separation of Security Domains

- Concerned about DoS attacks
  - If spaces are shared across tasks, how do we prevent even accidental DoS?
- Concerned about unauthorized access to data contained in a tuplespace
  - JavaSpace interface does not contemplate restricting access based on templates
- Solution: separate spaces for each Master/Worker session
- Each space to use a separate security domain
- Like a distributed ProtectionDomain

Dynamic Configuration

- Is what the “Flash” in Flashgridding means
- All services are “bootstrapped” via configuration URLs
  - Services can be remotely rebooted if necessary
- Variety of configuration languages can be used
  - We’ve put Groovy in our Jini based ConfigurationProvider
  - Jini based ConfigurationProvider using groovy included as part of ComputeCycles project
  - Worker services are configured in two phases
    - First as a worker service
    - Secondly when they participate in a grid session

Dynamic Allocation via Global Sessions as Leased Resources

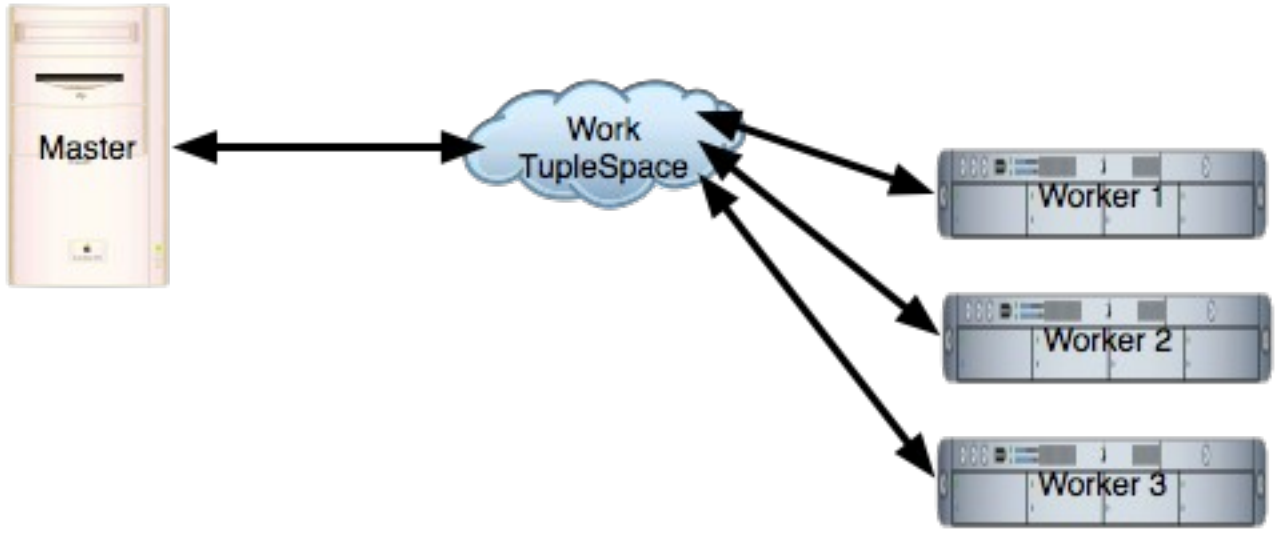
- Biggest problem we face is resource allocation and reclamation
- Copied idea of sessions from servlet containers
- There is a single leased resource in the grid: the session
  - Sessions are managed by a special service: the GridManager
  - Sessions reside in JavaSpaces and are visible to authorized services (such as available workers)
  - Workers participating in a session allocate themselves to that session dynamically and independently; they are only loosely coupled to the Master and are not directly under outside control

Master/Worker Pattern Usage

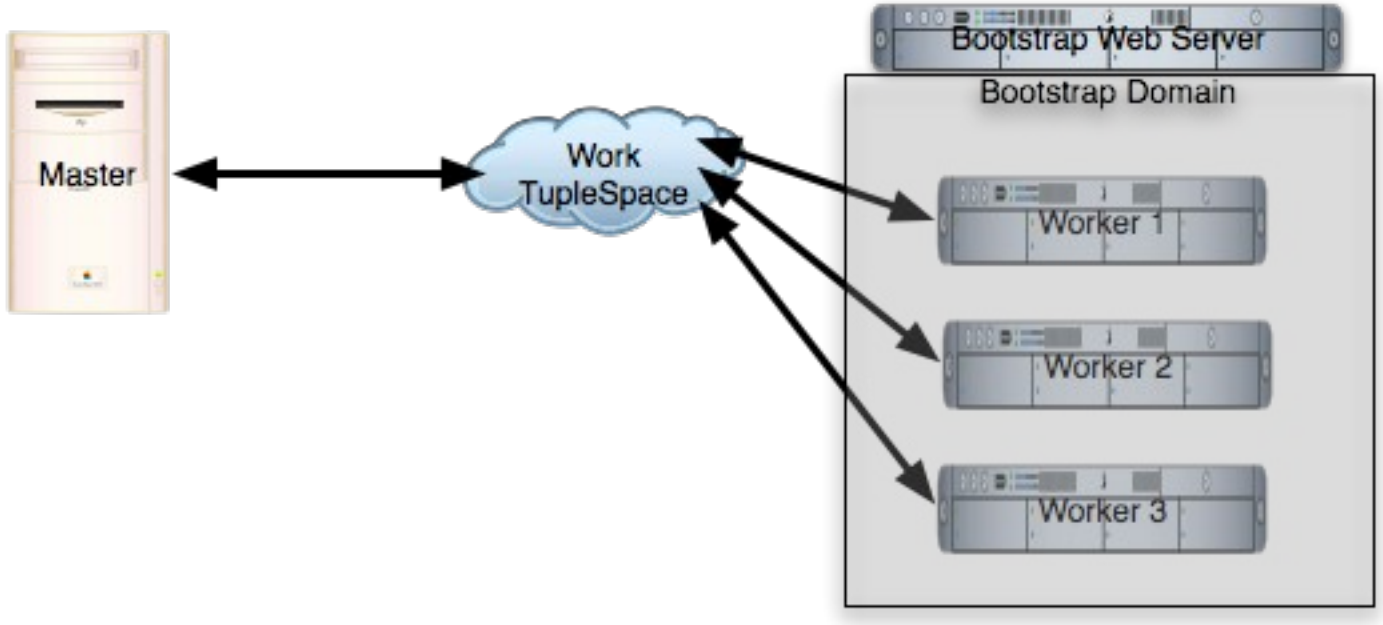
- Enhancing this pattern is the basis for ComputeCycles' contribution
- Key enhancements to remember
  - Global Sessions
  - Loose, dynamic coupling between masters and workers
- Masters can be WebServices
- Will eventually move to a workflow model
- Following diagrams show the actual evolution of the design (sadly)

ComputeCyles and the Master/Worker Pattern

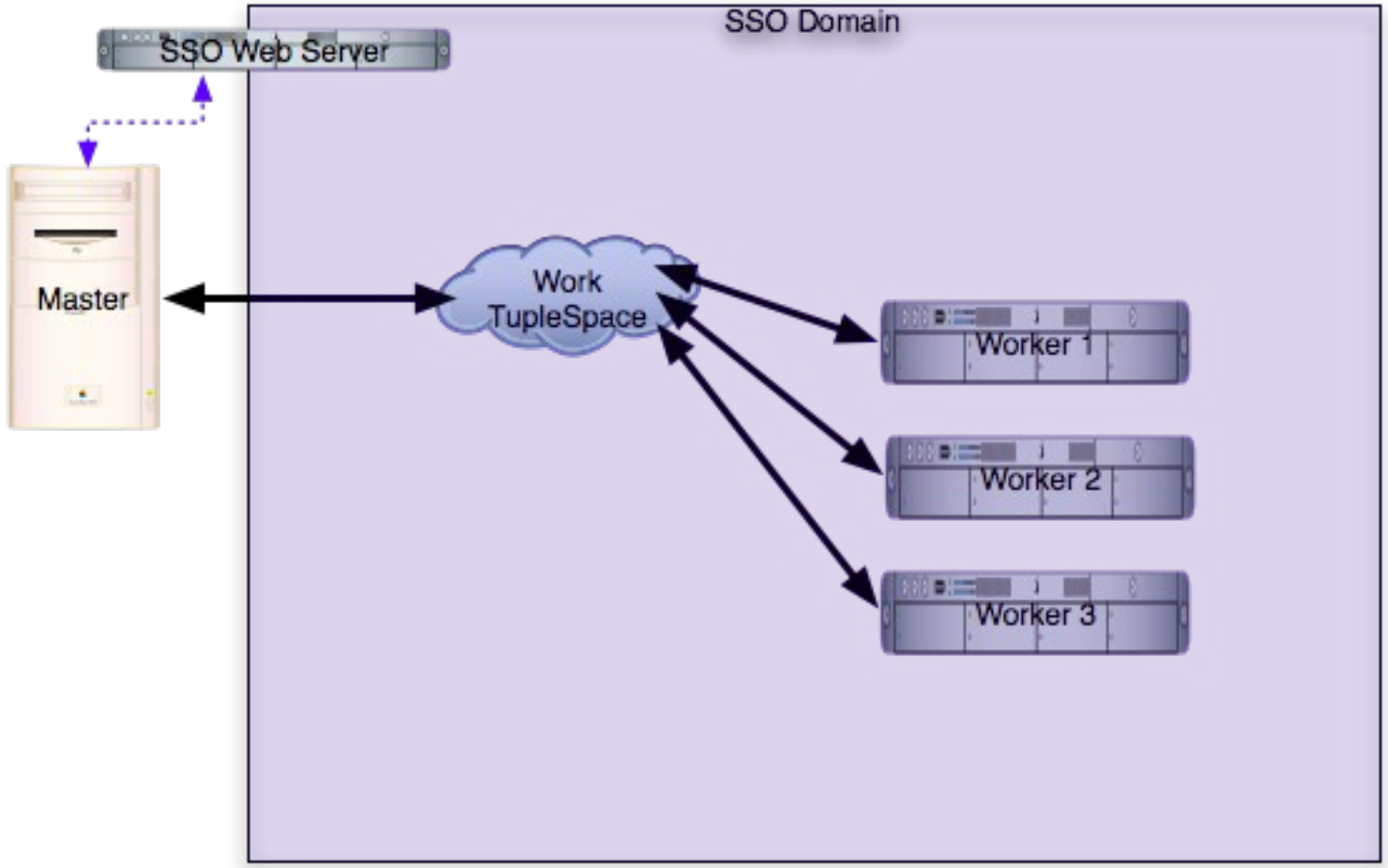
Turn away now if complication scares you…



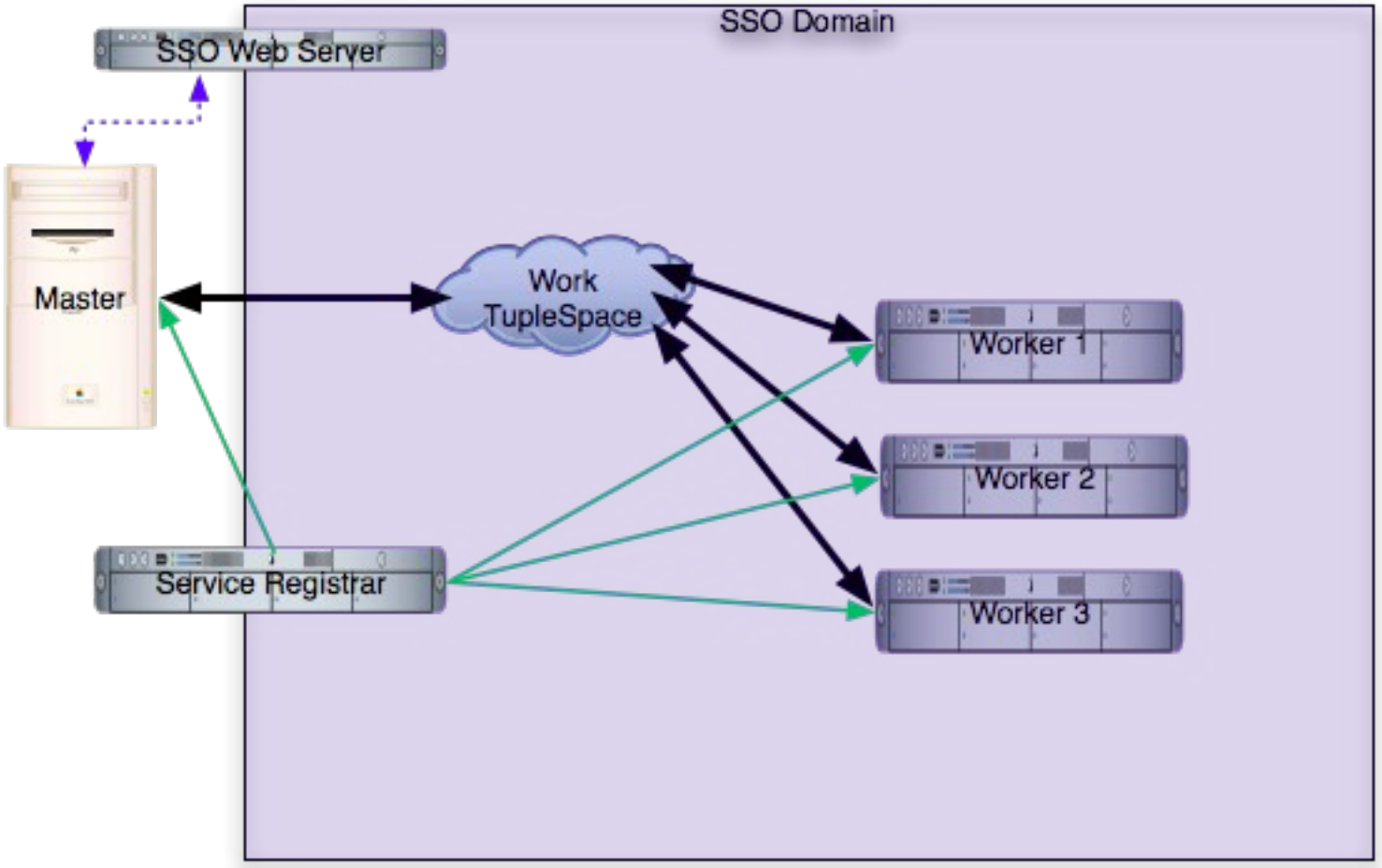
Bootstrapping Deployment



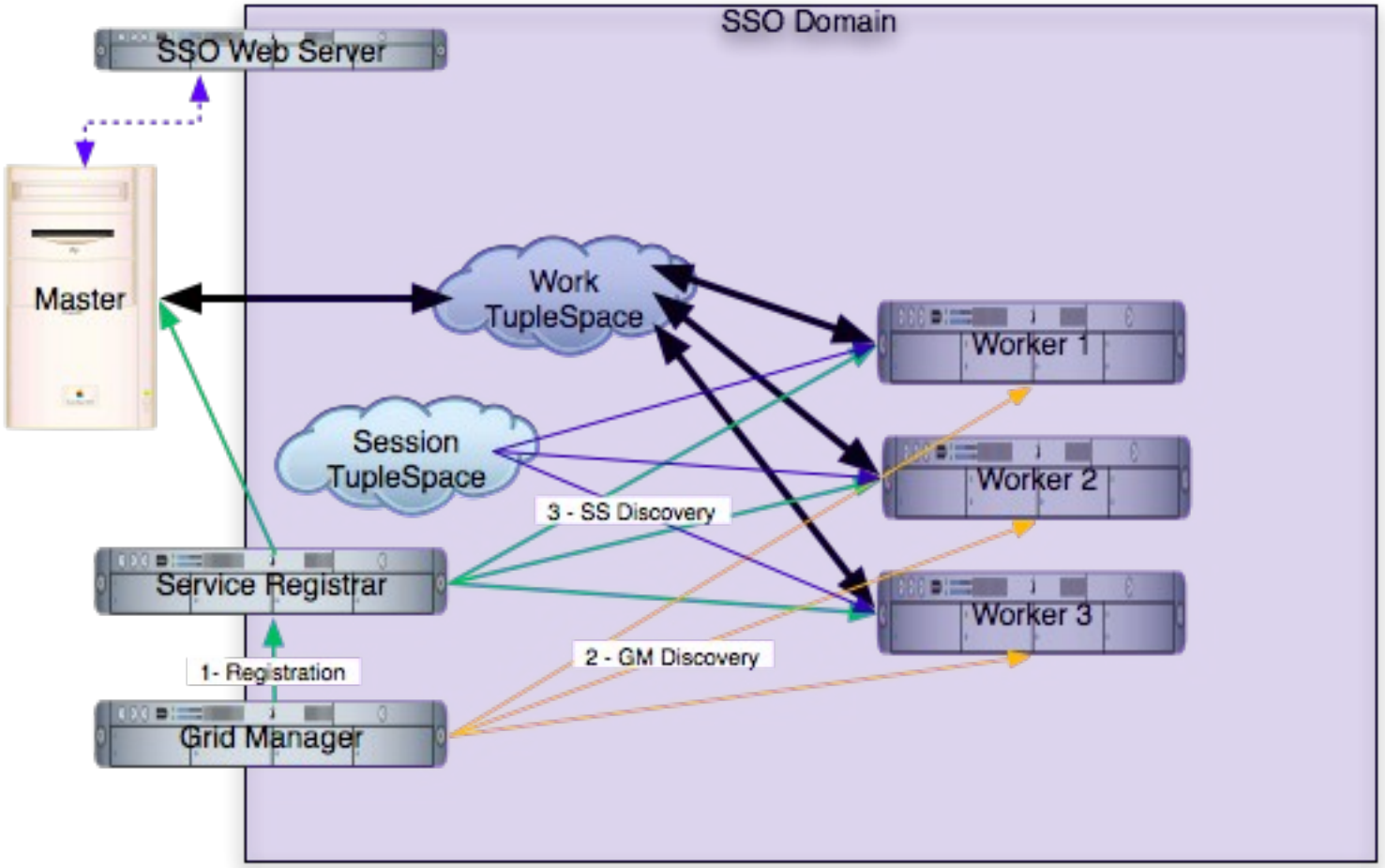
Single Sign-on Security



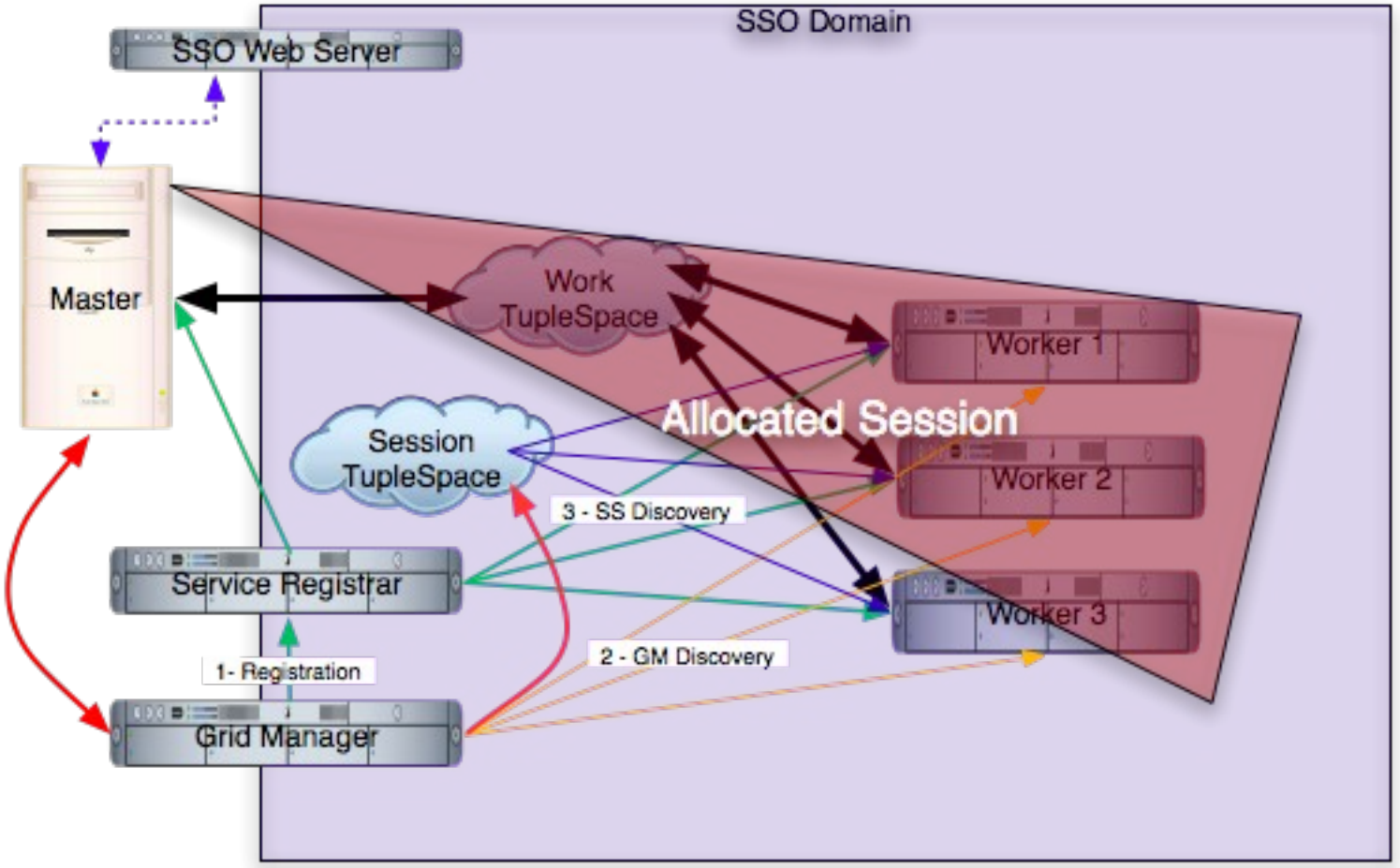
Service Registration



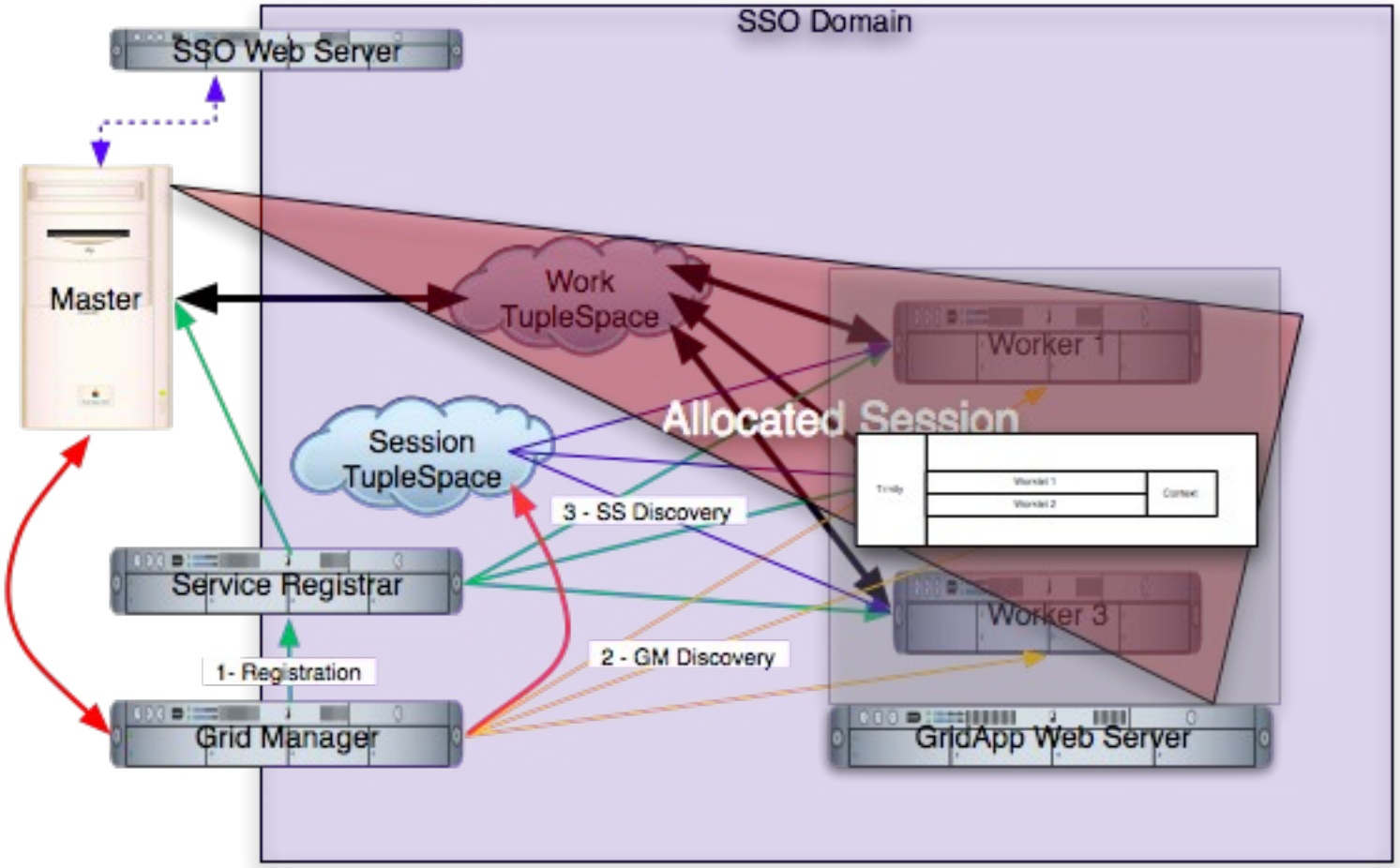
Grid Management



Session Allocation



Application Deployment



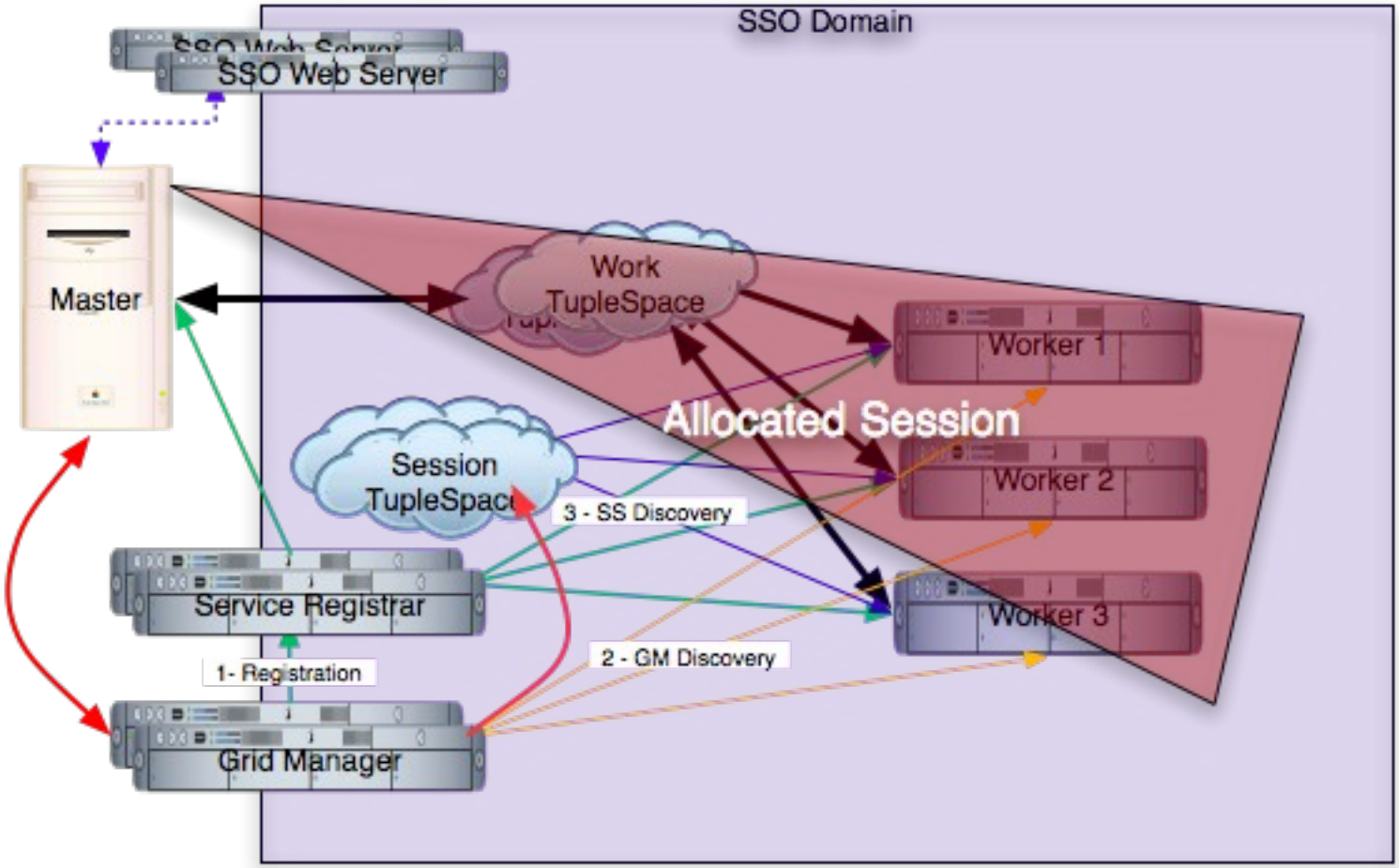
Zero Single-Point-of-Failure Architecture

- Puts the “Grid” in Flashgridding
- Requires the concept of a global, shared session and hence multiple tuple spaces
  - Sessions are leased resources which must be visible to workers
  - They must persist in multiple locations to avoid single-point-of-failure problems
  - Tuplespaces are ideal for this usage
- Replicating session entries across tuple spaces leads to a requirement for a transaction manager
  - Sessions are written to at least two locations while inside a transaction

Zero Single-Point-of-Failure Architecture (Cont.)

- Standard M/W pattern has this feature only on the workers, not the Masters
- GridManagers subject to same constraint
  - Teams of GridManager services renew leases
  - Any GridManager can fail and sessions that it originated can still be renewed by other members of the team

Zero Single-Point-of-Failure Requirements



Services

- Reggie (Lookup)
- Outrigger (JavaSpace)
- Mahalo (TxnManager)
- Trinity (Worker)
- Edison (GridManager)
- SpaceGhost (SpaceStarter)
- Pharoah (MasterDelegate)

Accessing the Flashgrid via WebServices

- Masters cannot be required to reside in a Java VM
  - WebService has been created to allow a master service to be started via remote invocation
- Jini Lookup Discovery cannot be performed through a firewall
  - Another WebService has been created to vend lookup proxies to remotely located services (e.g., workers)
  - Easiest mechanism is to store bootstrapped lookup service proxies in J.N.D.I. API and vend them over the webservice
  - Vended lookup proxies must use https w/o callbacks in order to go through firewalls

JavaSpace Usage

- JavaSpaces are used for all shared memory needs
- Types of spaces used
  - Session
  - Work (requires use of JavaSpace05 interface)
  - Scoreboarding
  - Master
- Session spaces are “infrastructural”, others are forked per session
- Each space must have a different LoginContext
  - Dynamic configuration of spaces is hard
  - We use our configuration webapp extensively here

Example: Simulation on Wall Street

- Fact: homeowners refinance fixed rate mortgages when rates drop
- Problem: how to value and hedge a pool of such mortgages
- Conventional solution
  - Simulate
    - Changes in rates
    - Refinancing behavior given rates
  - Calculate
    - Present value of cashflows along each simulation path
    - Hedge ratios (i.e., change in value per change in model parameter) by re-running simulation with slight change in initial parameters

Using a Grid for this Example

- Scale requirements:
  - 600 mortgage securities x 360 monthly cashflows x 2000 simulation paths x 20 scenarios = 8.64 Billion present values to calculate
  - This problem is “embarrassingly” parallel
- Example workflow
  - In the master create the simulation path parameters and place in space
  - In the worker, compute the paths from the params
  - In the master, repeatedly write (security,scenario) tuples into the space and wait for results
  - In the worker, repeatedly read, compute, write

Code Samples: What the Client Uses To Access the Flashgrid

```
public interface GridManager
{
    public ClientSession createSession(
                    String gridAppURL,
                    String gridAppUserName,
                    String gridAppPassword,
                    long sessionTimeout,
                    long waitTimeout)
        throws RemoteException,
               GridAppNotFoundException,
               NoSessionSpaceAvailableException,
               NoWorkSpaceAvailableException;

    public GridSessionStatus getSessionStatus(
                    GridSession gs)
        throws RemoteException;

}
```

Code Samples: What a Worker Looks Like to the Rest of the Grid

```
public interface Worker
    extends Remote
{
    // This has turned into a marker
    // interface as workers have become
    // independent agents
}
```

Code Samples: the Code That a GridApp Deploys into Each Worker

```
public interface Worklet
{
    public void destroy()
        throws WorkletException;

...

    public void init(WorkletDescriptor descriptor,
  WorkletContext context)
        throws WorkletException;

    public boolean isDispatchable()
        throws WorkletException;

    public void service(WorkletRequest req,
  WorkletResponse res)
        throws WorkletException;
}
```

Summary of ComputeCycles Project

- Open source
- Uses an infrastructure based on other open source projects
- Avoids single points of failure
- Uses the Java Security model (via Jini technology and Project GlassFish) throughout
- Is designed to be easy to deploy
- Wants to be a grid-based workflow engine when it grows up

For More Information

http//:computecycles.dev.java.net

http//:computeserver.dev.java.net

Q&A

Flashgridding with Java: Using Project GlassFish℠, JavaSpaces™ and Groovy in an Open Source Supercomputer

Van Simmons, Sean Merritt and Jim Gammill

Project Leaders
ComputeCycles Project
http://computecycles.dev.java.net

TS-3714