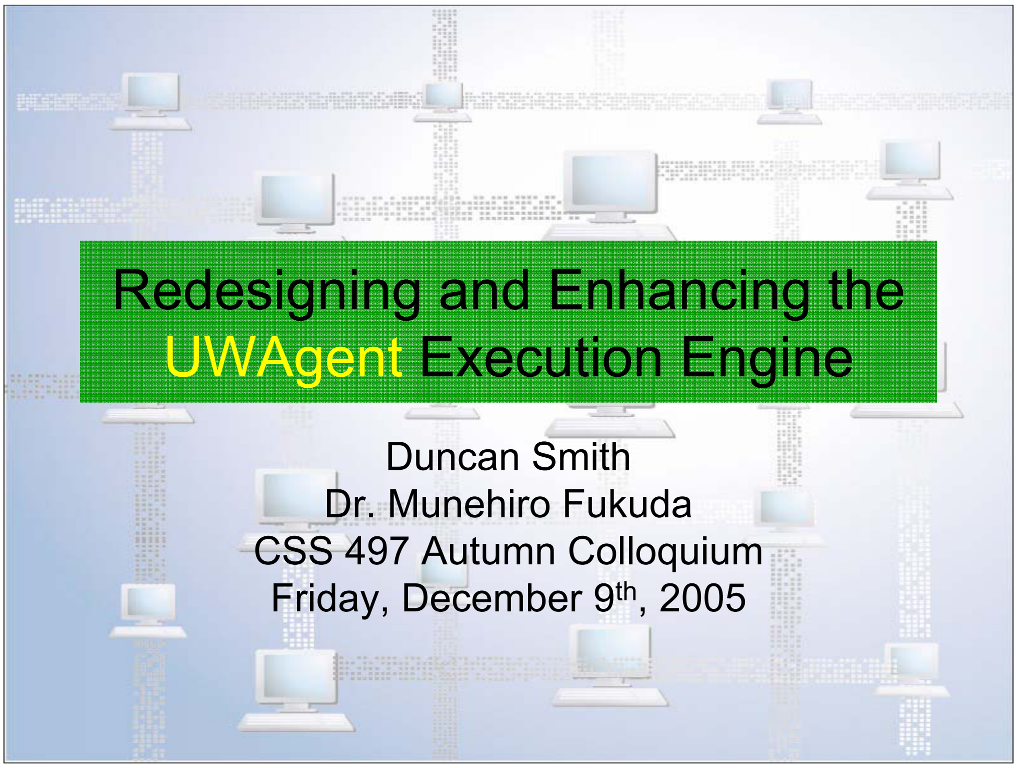
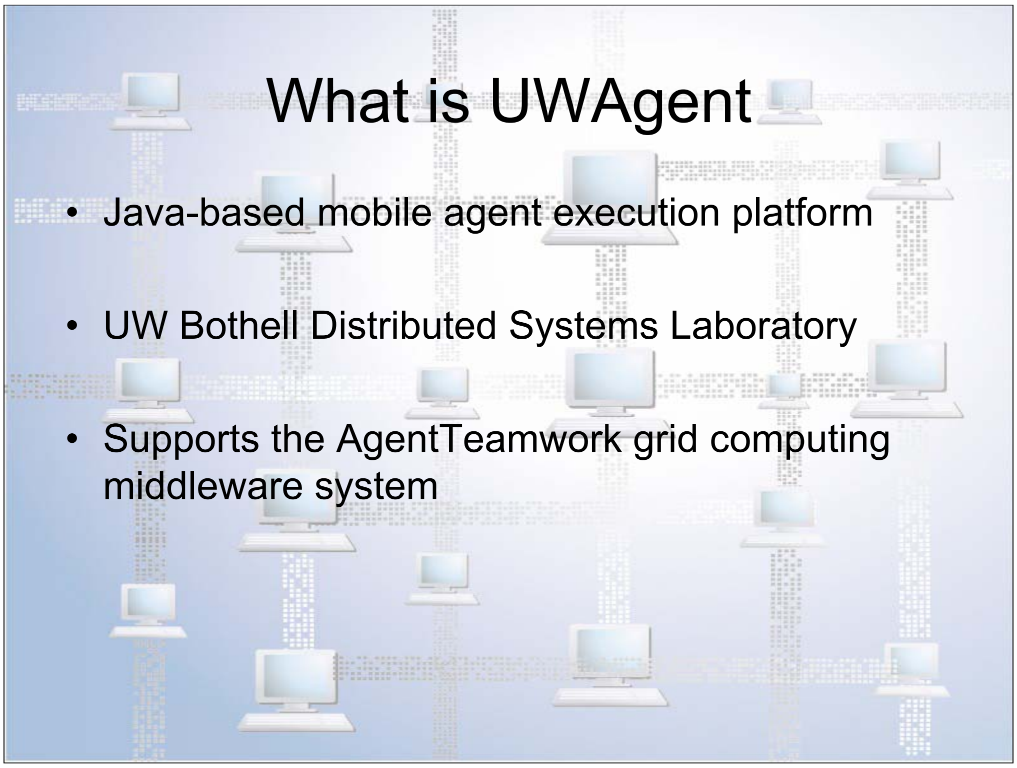


Redesigning and Enhancing the UWAgent Execution Engine

Duncan Smith
Dr. Munehiro Fukuda
CSS 497 Autumn Colloquium
Friday, December 9th, 2005

10-15 minutes

Thanks for coming, everyone. I’m Duncan Smith, and for the past 5 months I have been working with Professor Fukuda on the UWAgent system.



What is UWAgent

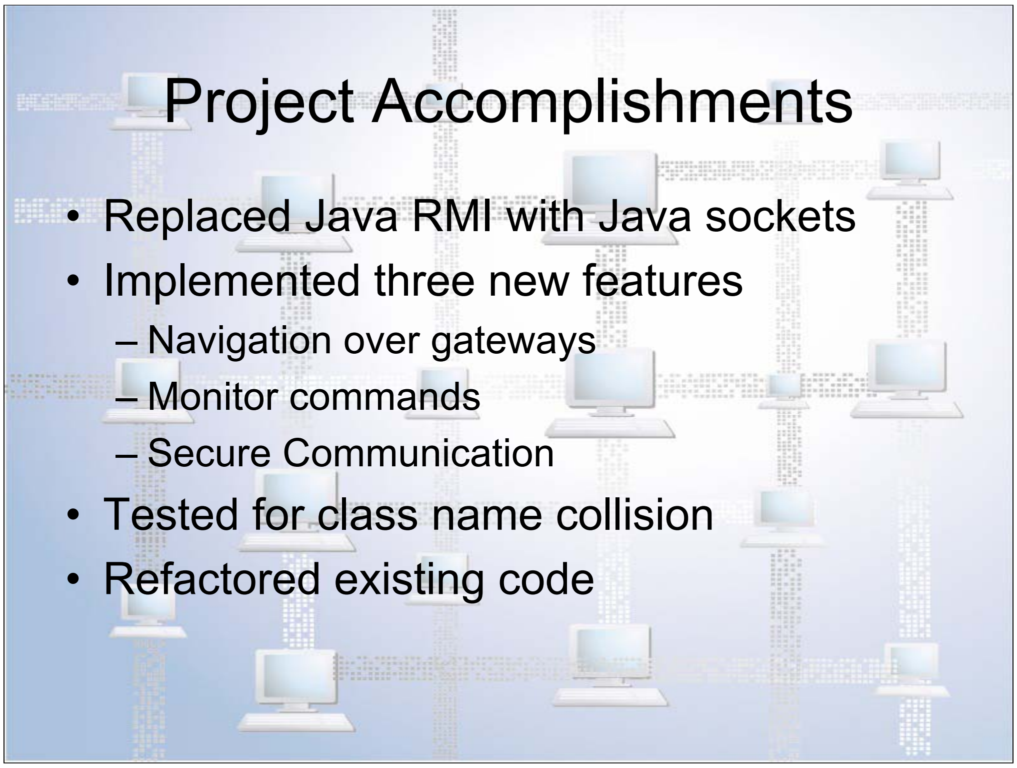
- Java-based mobile agent execution platform
- UW Bothell Distributed Systems Laboratory
- Supports the AgentTeamwork grid computing middleware system

Some of you may have heard of UWAgent, but for those who haven't, here is a quick overview. UWAgent is a Java-based mobile agent execution platform. A mobile agent is a program that can move from one computer to another autonomously. For example, if you wanted to buy an airline ticket, you might use a mobile agent that would visit a number of different travel sites to compare prices. In a mobile agent environment, you wouldn't need to pull itinerary data over the Internet to your local machine for processing. Instead, your agent could do all of its processing at the remote site, and return with only the results.

UWAgent has been developed over the years by students and faculty working at the UW Bothell Distributed Systems laboratory. Currently, UWAgent is being used to support the AgentTeamwork grid computing middleware system. However, it is written so that it could be used by any system needing mobile agent support. One of the tasks from my work these past two quarters has been to remove AgentTeamwork-specific code from the UWAgent code base.

AgentTeamwork People

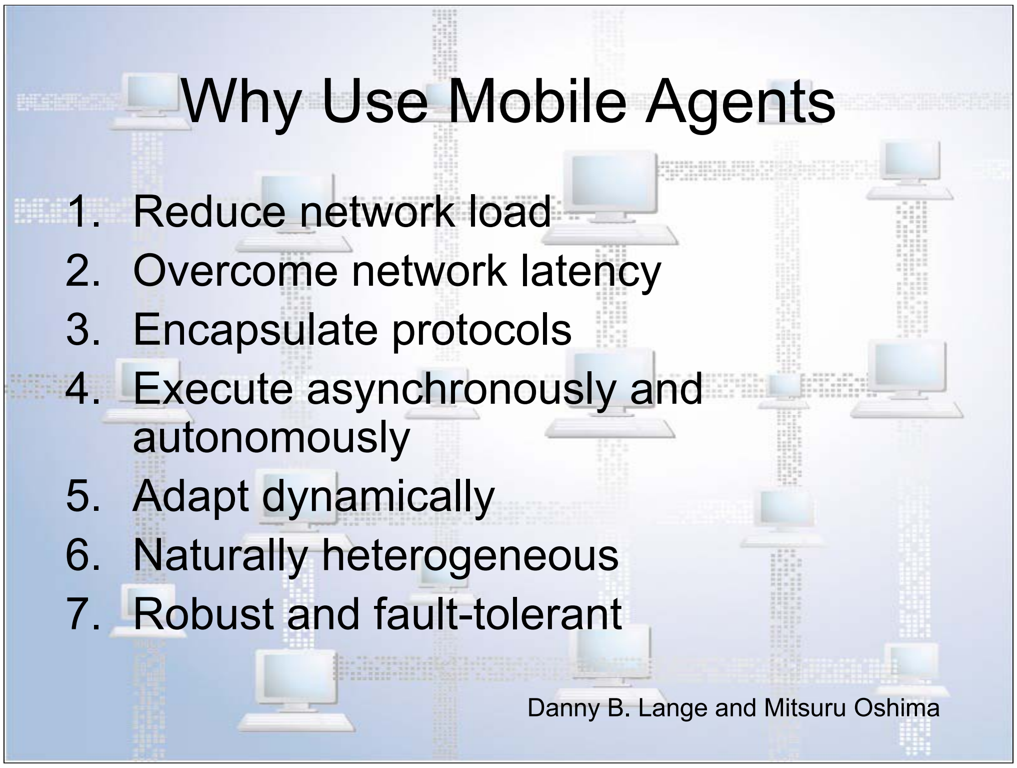
These are the people who have worked on UWAgent and AgentTeamwork over the years.



Project Accomplishments

- Replaced Java RMI with Java sockets
- Implemented three new features
  - Navigation over gateways
  - Monitor commands
  - Secure Communication
- Tested for class name collision
- Refactored existing code

Here are the things we did during this part of the UWAgent project. I’ll talk about each of these individually during this presentation. We replaced Java RMI with Java sockets. We implemented three new features. We verified that UWAgent isn’t affected by a potential problem common to mobile agent systems. And we did some code cleanup.

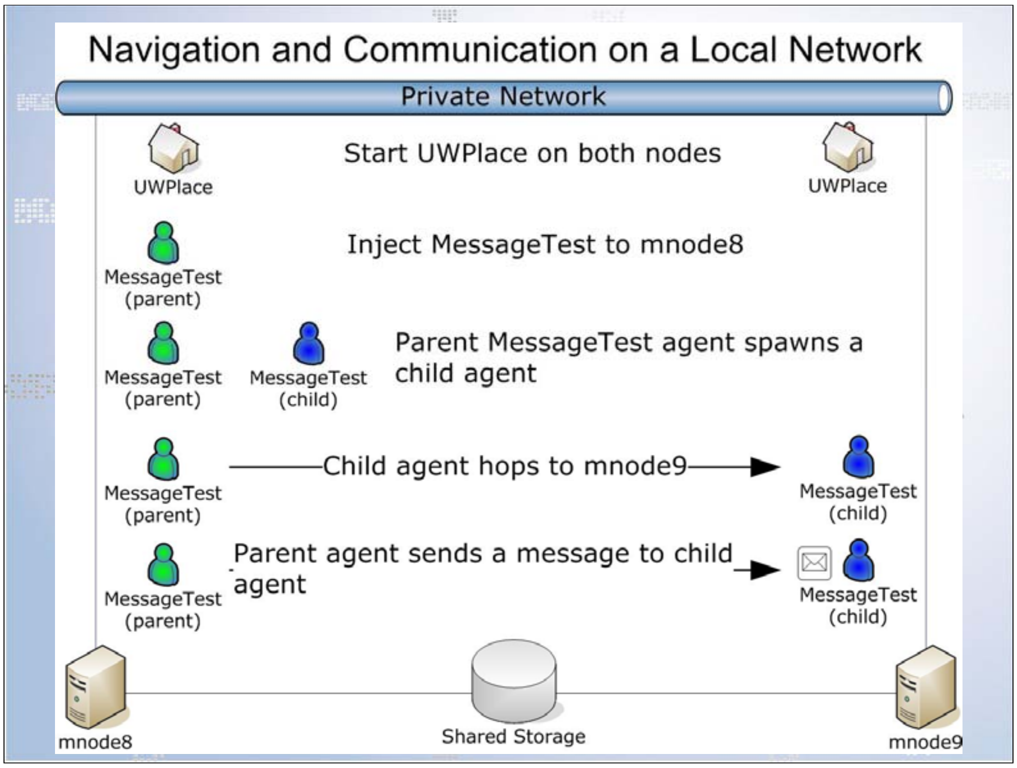


Why Use Mobile Agents

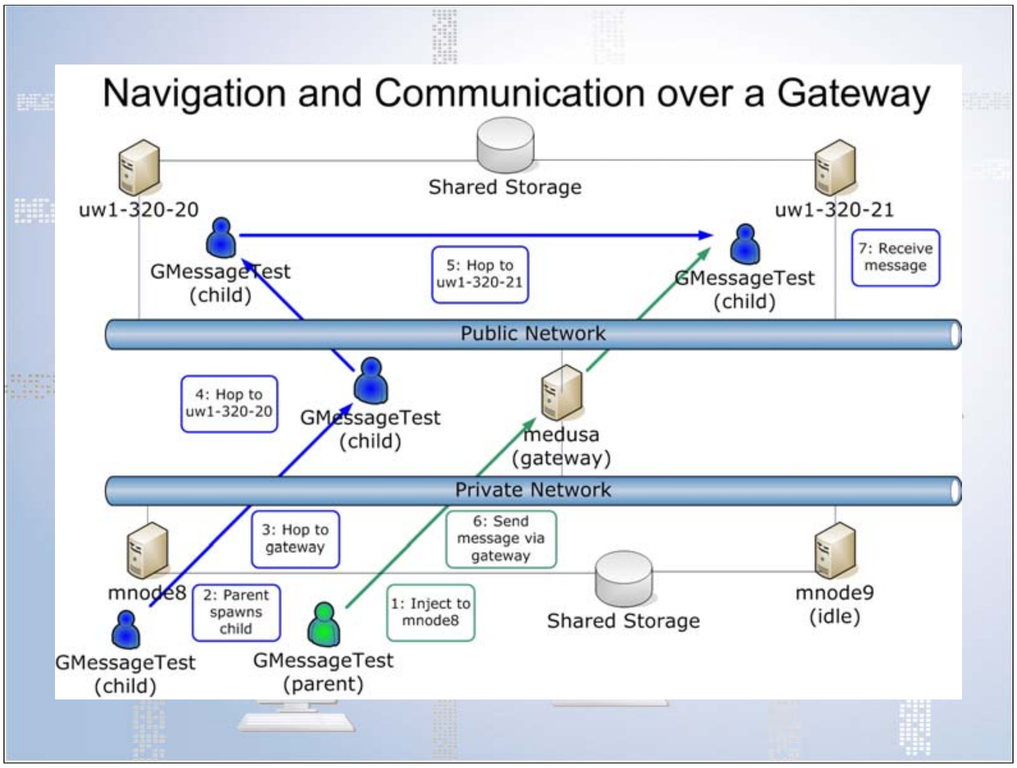
1. Reduce network load
2. Overcome network latency
3. Encapsulate protocols
4. Execute asynchronously and autonomously
5. Adapt dynamically
6. Naturally heterogeneous
7. Robust and fault-tolerant

Danny B. Lange and Mitsuru Oshima

Before I talk more about UWAgent specifically, here is a list of seven reasons to use mobile agents. The list comes from Danny Lange and Mitsuru Oshima, two researchers who developed another Java-based mobile agent system called Aglets. They say that mobile agents reduce network load, because they process data locally rather than pulling it over the network. They overcome network latency, for the same reason. They encapsulate protocols, meaning that agents rather than hosts can take responsibility for implementing protocols, which may be obscure and only applicable to the task the agent is doing. They execute asynchronously and autonomously. They adapt dynamically to changing conditions on the host (such as increasing load from other processes). They are naturally heterogeneous because they only depend on their execution environment, not the underlying host. And they are robust and fault-tolerant, as a result of the previous six reasons.

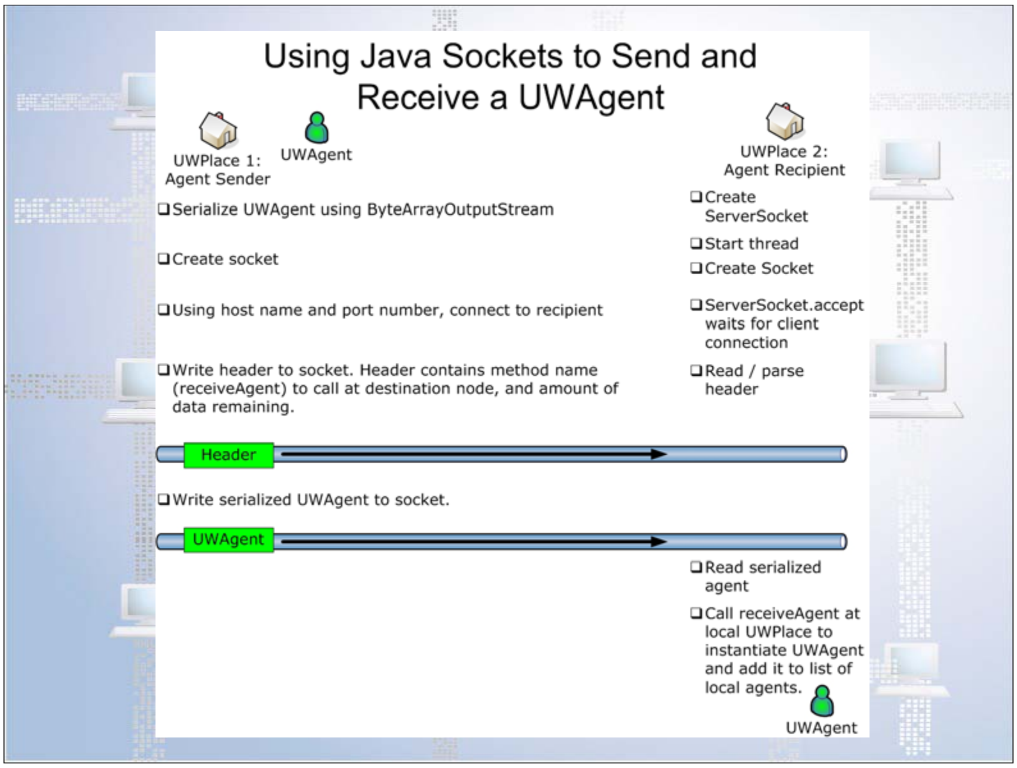

Navigation and Communication on a Local Network
Private Network
UWPlace
Start UWPlace on both nodes
UWPlace
MessageTest (parent)
Inject MessageTest to mnode8
MessageTest (parent)
MessageTest (child)
Parent MessageTest agent spawns a child agent
MessageTest (parent)
Child agent hops to mnode9
MessageTest (child)
MessageTest (parent)
Parent agent sends a message to child agent
MessageTest (child)
mnode8
Shared Storage
mnode9

This diagram shows how UWAgents move and communicate when their hosts are on a single network. The bar at the top represents the network, the houses are the two hosts that the agents execute on, and the drum at the bottom is the disk storage that all of the hosts on the network share. That is a useful feature in a mobile agent system, because it reduces the amount of information that agents need to carry with them. At the top, we have a UWPlace on each node. This is the execution environment that must be running on each node that a UWAgent uses. Next we inject an agent to one of the nodes, meaning we start its execution there. The agent then spawns a child agent, meaning that it creates an autonomous copy of itself. The child agent then moves, or “hops” to another node. Finally, the parent agent sends its child a message. The UWAgent system facilitates this message delivery by keeping track of where each agent is located.



Navigation and Communication over a Gateway

This diagram shows a similar process to the previous one, except now we have two networks that are connected by a gateway. Again, an agent is injected to one of the nodes, and it spawns a child on the same node. However, the child now wants to hop to a machine on a different network. In order to do this, it needs to take a detour through the gateway machine, which is also running the UWAgent system. It then hops out to the public network, and finally to another machine on the same network. The parent then sends a message. The UWAgent system has kept track of the location of the child agent through all of its hops, so it knows which machine the child is on. Because the destination machine is not visible from the private network, the message also needs to detour through the gateway machine before it can reach its destination agent.



Using Java Sockets to Send and Receive a UWAgent

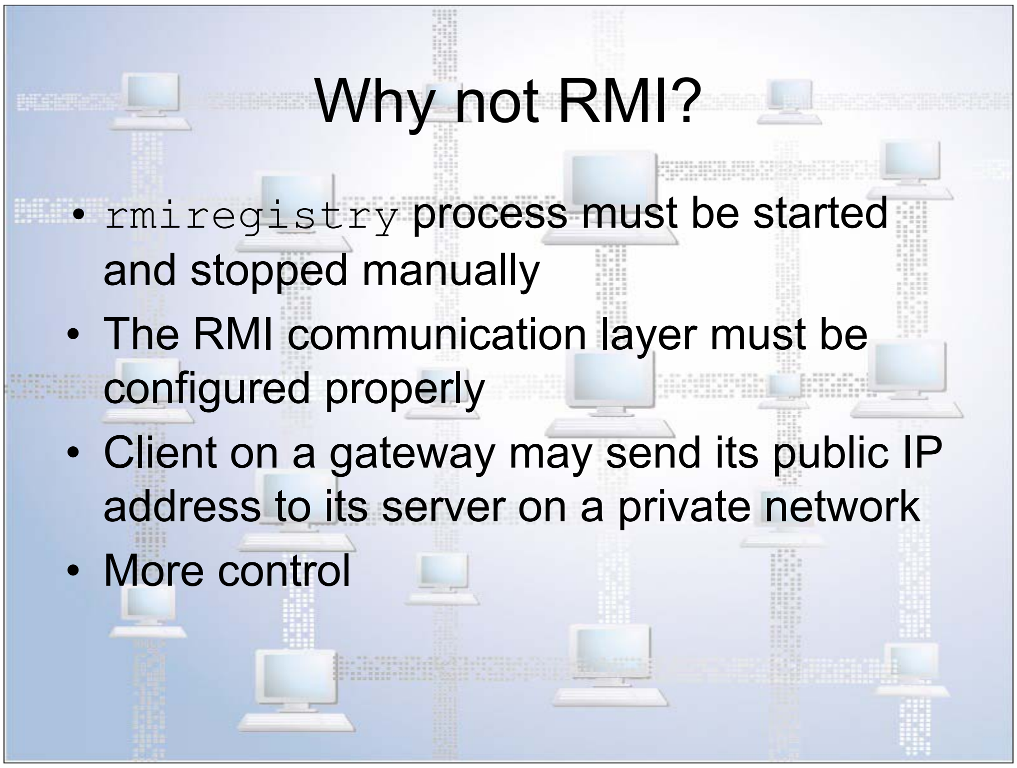
UWPlace 1: Agent Sender

- Serialize UWAgent using ByteArrayOutputStream
- Create socket
- Using host name and port number, connect to recipient
- Write header to socket. Header contains method name (receiveAgent) to call at destination node, and amount of data remaining.
- Write serialized UWAgent to socket.

UWPlace 2: Agent Recipient

- Create ServerSocket
- Start thread
- Create Socket
- ServerSocket.accept waits for client connection
- Read / parse header
- Read serialized agent
- Call receiveAgent at local UWPlace to instantiate UWAgent and add it to list of local agents.

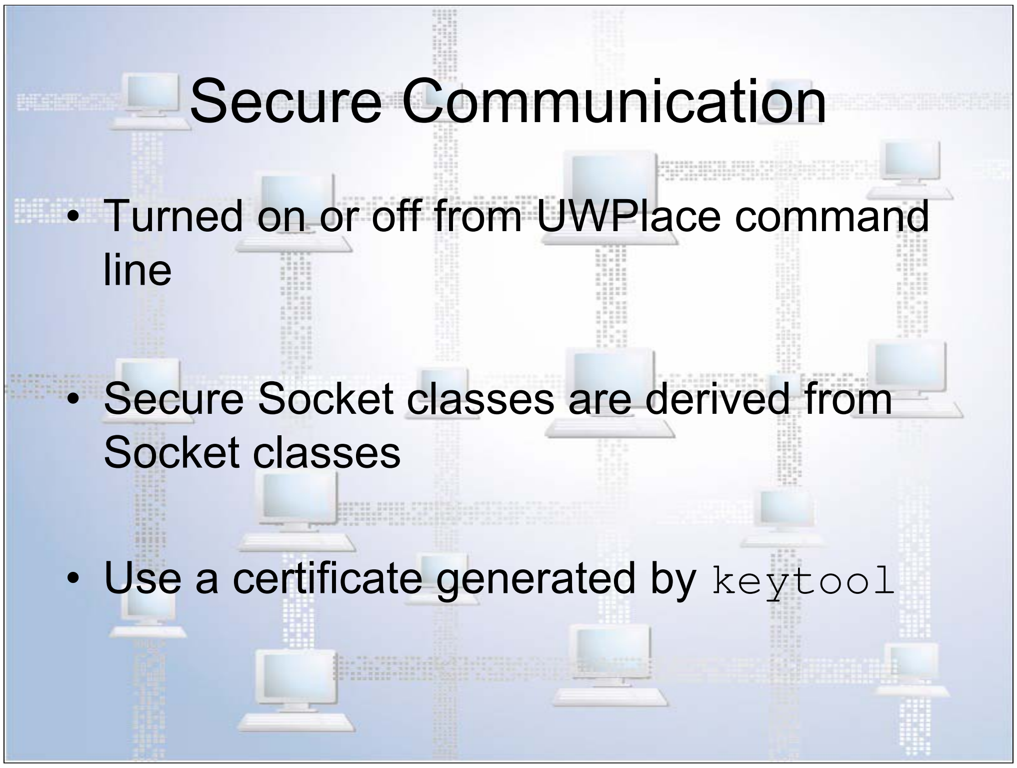
This diagram shows some lower-level mechanics about how messages are sent between nodes using our custom socket interface. This applies to all remote method calls in the UWAgent system. These remote method calls were handled by RMI in the previous version of the system. In this example, I'll use the receiveAgent method, which is part of the agent navigation process. On the left side, we have a UWPlace with one UWAgent. The agent wants to hop to this other UWPlace on the right hand side. The first step is to serialize the agent. This puts it in a form where it can be transferred over the network. Meanwhile, the recipient (along with every UWPlace) has created a ServerSocket to receive incoming requests. It has also started a separate thread to listen for requests and a socket to be used when a request comes in. The sender also creates a socket, and connects to the recipient, which accepts the connection. The sender writes a fixed-length header to the socket. The header contains the method name (receiveAgent, in this case), and the amount of data to be sent. This second piece of information will be crucial at the destination, because of the way Java sockets work. In fact, it was probably the trickiest part of the enhancement work. The recipient reads and parses the header, and the sender writes the serialized agent to the socket. The recipient reads the correct number of bytes to get the agent. It then calls the specified method, receiveAgent, passing in a byte array. receiveAgent instantiates the agent using its serialized form, and adds it to the list of local agents, where it will receive its share of the local resources.



Why not RMI?

- `rmiregistry` process must be started and stopped manually
- The RMI communication layer must be configured properly
- Client on a gateway may send its public IP address to its server on a private network
- More control

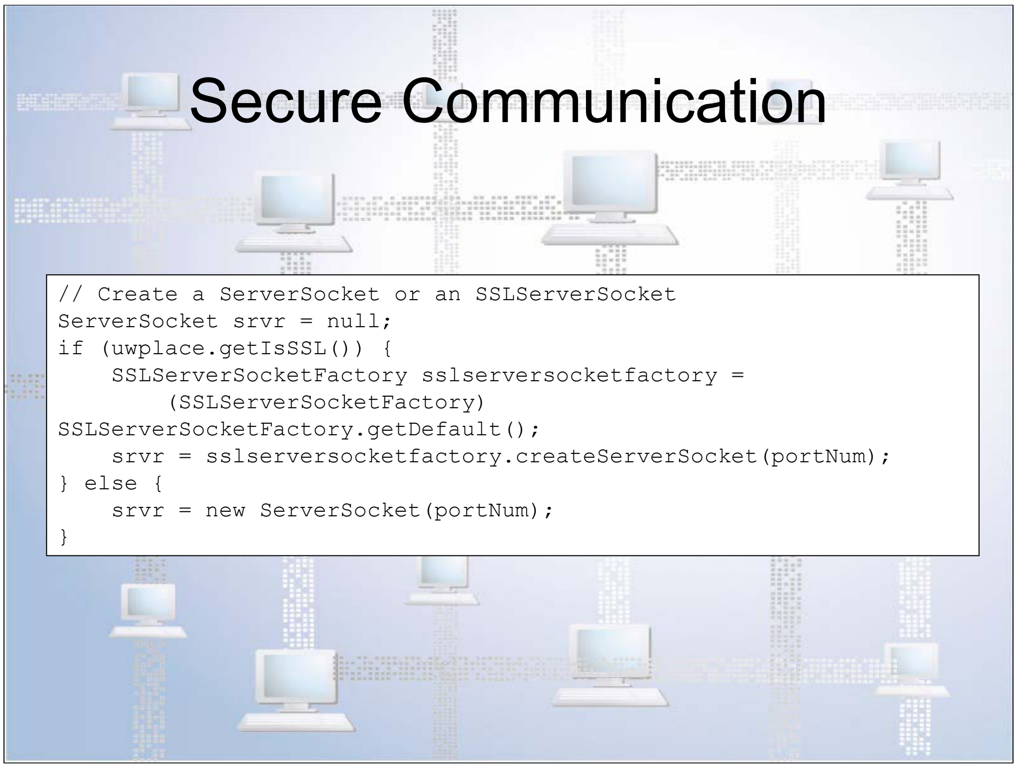
After learning about the socket process, you may be wondering why we didn't just stick with RMI rather than inventing our own process. Here are a few reasons. First, there is a program called rmiregistry that must be running on each node. It must be started and stopped manually with the correct port number. RMI introduces a communication layer that must be configured properly. A potential problem exists when a client is on a gateway machine and it establishes a connection to a server on the private network side of the gateway. In some cases, the client sends the gateway's public IP address rather than the private one. Finally, a custom socket interface gives us more control, which can be useful in a research environment.



Secure Communication

- Turned on or off from UWPlace command line
- Secure Socket classes are derived from Socket classes
- Use a certificate generated by `keytool`

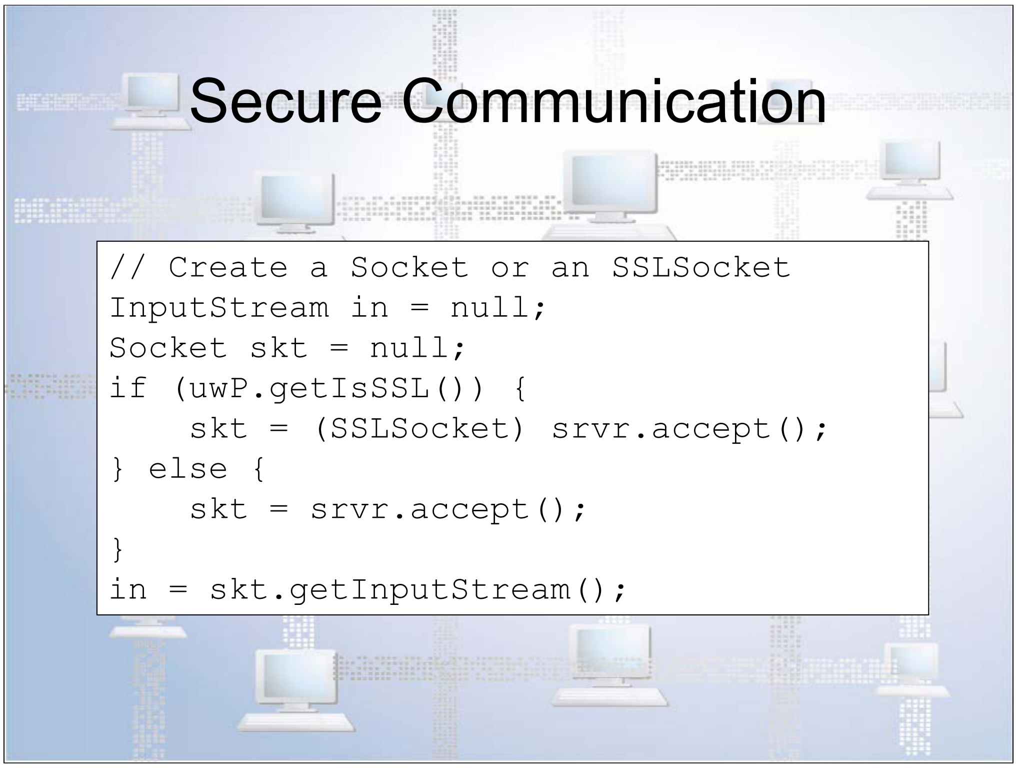
One of the ways we used this enhanced control over the remote method call process is to provide secure communication between UWPlaces. This security support is optional, so the user can turn it on and off at the UWPlace command line. Of course, the settings must match between UWPlaces. It was implemented using Java's SSLSocket and SSLServerSocket, as we'll see in a minute. And it uses a certificate generated by the keytool command, which is part of the Java SDK.



Secure Communication

```
// Create a ServerSocket or an SSLServerSocket
ServerSocket srvr = null;
if (uwplace.getIsSSL()) {
    SSLServerSocketFactory sslserversocketfactory =
        (SSLServerSocketFactory)
SSLServerSocketFactory.getDefault();
    srvr = sslserversocketfactory.createServerSocket(portNum);
} else {
    srvr = new ServerSocket(portNum);
}
```

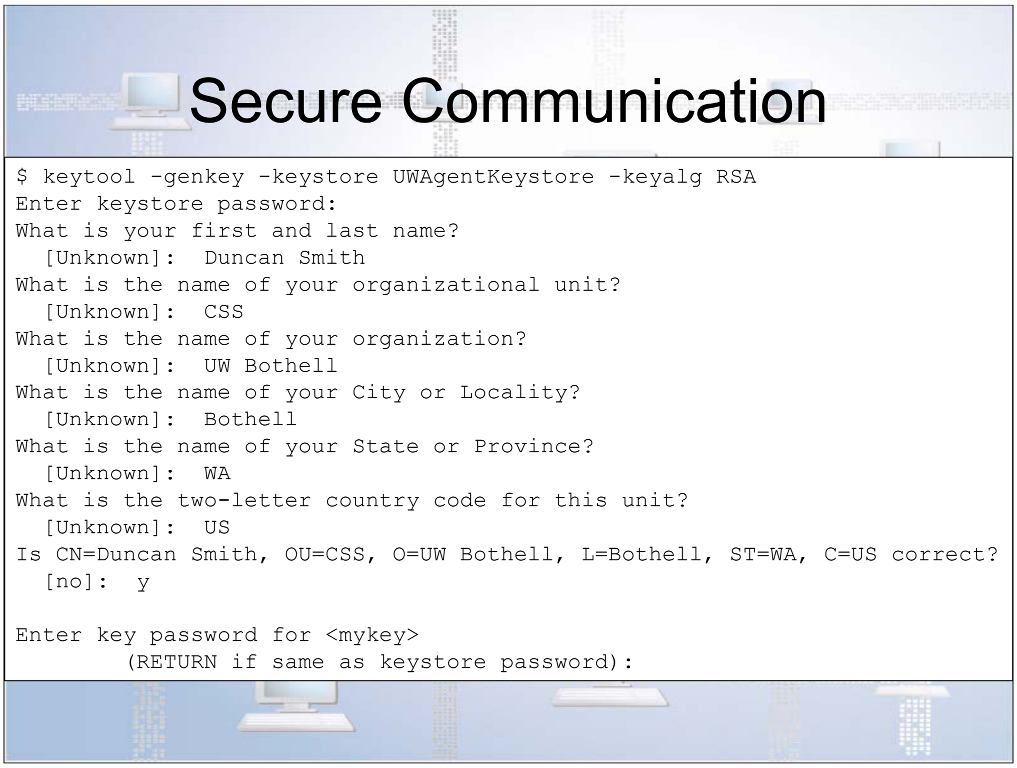
Here’s some code showing how a secure or a regular ServerSocket are created. I was pleased to learn that Java’s SSLServerSocket is derived from the base ServerSocket class. Because of this, I could use a base class variable to hold either socket type. Therefore, the code that checks the security option can be isolated to this part of the program, and the rest of the code can simply use this ServerSocket variable.



Secure Communication

```
// Create a Socket or an SSLSocket
InputStream in = null;
Socket skt = null;
if (uwP.getIsSSL()) {
    skt = (SSLSocket) srvr.accept();
} else {
    skt = srvr.accept();
}
in = skt.getInputStream();
```

Here’s the equivalent code for the client socket. The InputStream is used to retrieve the data coming over the socket. Again, we can use an instance of the base class to store either a regular or a secure socket.

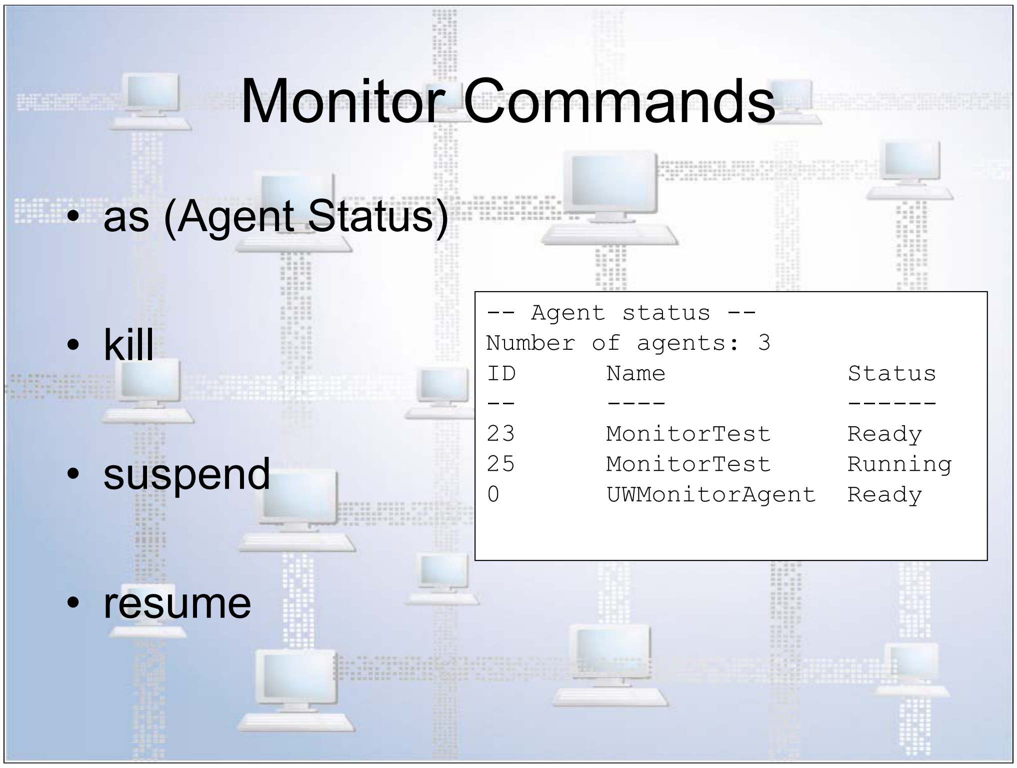


Secure Communication

```
$ keytool -genkey -keystore UWAgentKeystore -keyalg RSA
Enter keystore password:
What is your first and last name?
  [Unknown]:  Duncan Smith
What is the name of your organizational unit?
  [Unknown]:  CSS
What is the name of your organization?
  [Unknown]:  UW Bothell
What is the name of your City or Locality?
  [Unknown]:  Bothell
What is the name of your State or Province?
  [Unknown]:  WA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Duncan Smith, OU=CSS, O=UW Bothell, L=Bothell, ST=WA, C=US correct?
  [no]:  y

Enter key password for <mykey>
        (RETURN if same as keystore password):
```

And here’s the certificate generation process.

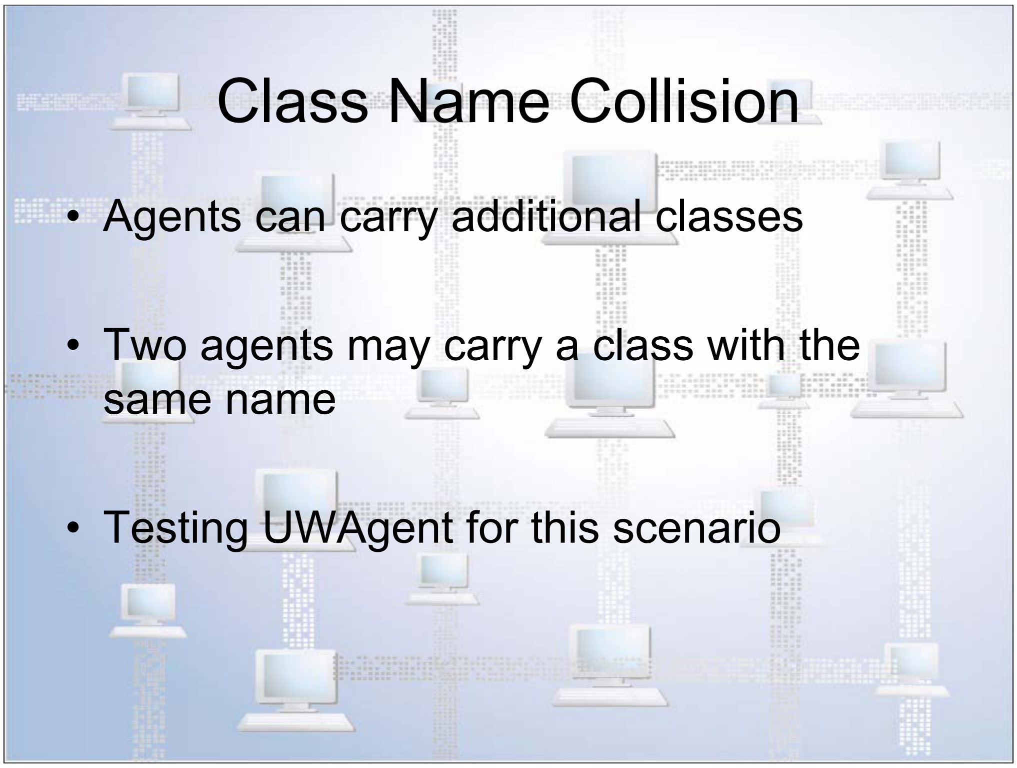


Monitor Commands

- as (Agent Status)
- kill
- suspend
- resume

```
-- Agent status --
Number of agents: 3
ID      Name            Status
--      ----            ------
23      MonitorTest     Ready
25      MonitorTest     Running
0       UWMonitorAgent  Ready
```

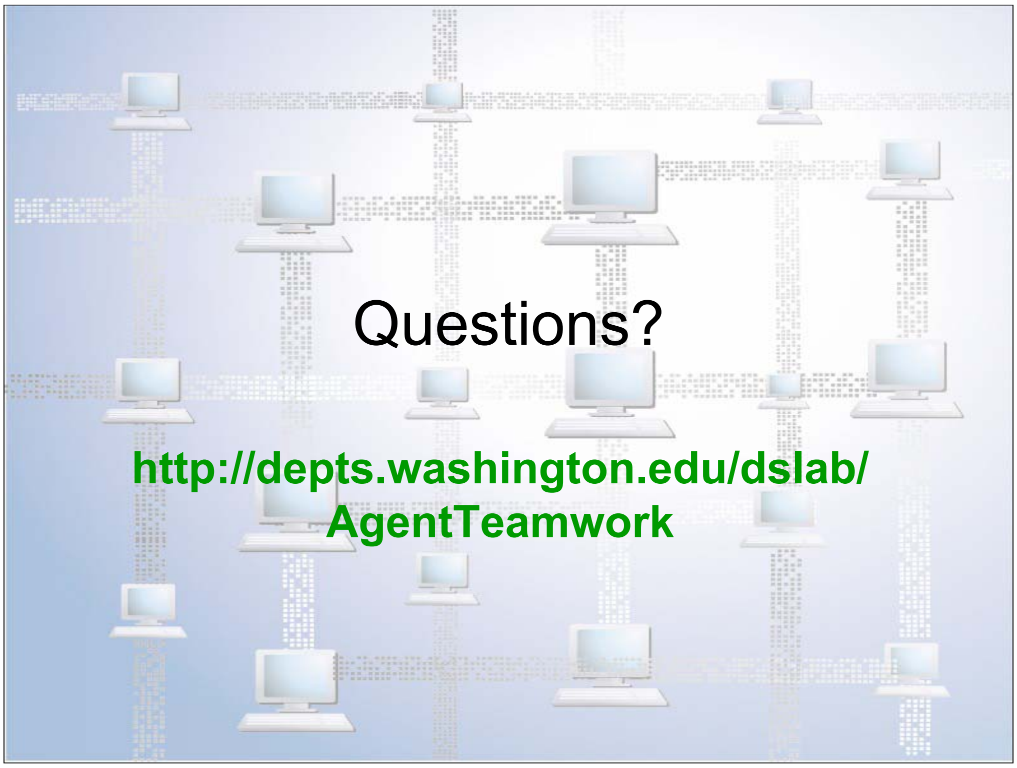
I'll conclude with one more feature, as well as one feature that I didn't have to implement. The feature that did get implemented is a set of four monitor commands, to provide better control over agents during experiments. Here's an example of the output from that command.



Class Name Collision

- Agents can carry additional classes
- Two agents may carry a class with the same name
- Testing UWAgent for this scenario

And the last feature is something we thought might be necessary, but turned out not to be. UWAgent allows agents to carry supporting classes with them when they hop from place to place. It is possible that two agents may each carry their own copy of a class, and that those classes may have the same name and same interface, but different behavior. One example is two students who may be running their distributed computing assignments on the same node. In systems that implement caching, such as IBM Aglets, there is a risk that the wrong method may be called. Aglets solves this problem by using separate class loaders that isolate the identically-named classes from each other. In the case of UWAgent, it turns out that Java protects against this problem automatically. We did a number of tests to confirm this.



Questions?

http://depts.washington.edu/dslab/AgentTeamwork

Thanks again for coming, everyone. Are there any questions?