UWAgents User’s Manual
(version 1.01)

Munehiro Fukuda ¹  Koichi Kashiwagi ²  Eric Nelson ¹,²
Duncan Smith ¹

¹ Computing & Software Systems, University of Washington, Bothell,
² Computer Science, Ehime University,
## Contents

1 UWAgents System ........................................... 3

2 UWPlace .................................................. 4
   2.1 Agent Scheduling ...................................... 5

3 UWInject .................................................. 6

4 UWAgent .................................................... 7
   4.1 Agent ID ................................................... 7
   4.2 Agent Programming ..................................... 7
   4.3 Postponed and Cascading Termination ................. 8
   4.4 Methods .................................................. 8

5 Inter-agent communication .................................. 10
   5.1 Message-Sending Procedure ............................ 10
   5.2 Message-Receiving Procedure .......................... 10
   5.3 UWMessage Methods .................................... 10

6 Credits ..................................................... 11
   6.1 Faculty ................................................... 11
   6.2 Collaborators ............................................ 11
   6.3 Students ................................................ 12

7 Final Comments ........................................... 12
1 UWAgents System

The UWAgents system is a Java-based mobile-agent execution platform that is being developed by the Distributed Systems Laboratory at UW Bothell. It is particularly intended for (but not limited to) the underlying infrastructure of our mobile-agent-based PC grid environment, AgentTeamwork. For this purpose, the system will be continuously enhanced for improved inter-agent communication and security.

The two key components in the UWAgents system are UWAgent and UWPlace. The former is a mobile agent that coordinates job executions on behalf of its client user. The latter is a place where agents migrate autonomously, work for their clients, and interact with each other.

The UWPlace component consists of the AgentThread, CommandLineOptionAnalyzer, UWClassLoader, UWObjectInputStream, UWPlace, and UWTransitSystem classes. The UWAgent component consists of the UWAgent, UWAgentMailbox, UWMessagingSystem classes.

Table 1 provides a brief description of these and other classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentThread</td>
<td>Execute the specified method of an incoming UWAgent.</td>
</tr>
<tr>
<td>CommandLineOptionAnalyzer</td>
<td>Parse the UWPlace and UWInject command line.</td>
</tr>
<tr>
<td>UWClassLoader</td>
<td>Instantiate a class from a byte array in memory.</td>
</tr>
<tr>
<td>UWObjectInputStream</td>
<td>Instantiate a class from a byte array created by ObjectOutputStream.</td>
</tr>
<tr>
<td>UWPlace</td>
<td>The place itself.</td>
</tr>
<tr>
<td>UWTransitSystem</td>
<td>The UWPlace interface.</td>
</tr>
<tr>
<td>UWAgent</td>
<td>Abstract class. Extend UWAgent to create a user agent.</td>
</tr>
<tr>
<td>UWAgentMailbox</td>
<td>Allows UWAgents to communicate with each other through RMI.</td>
</tr>
<tr>
<td>UWMessagingSystem</td>
<td>The message that is transmitted from AgentMailbox to AgentMailbox when UWAgents communicate. Uses String [] for the message header and Hashtable for the message.</td>
</tr>
<tr>
<td>UWInject</td>
<td>Inject an agent onto a UWPlace. (See sec. 3).</td>
</tr>
<tr>
<td>AgentUtil</td>
<td>Contains common processing for CommanderAgent, SentinelAgent and BackupAgent (parts of AgentTeamwork).</td>
</tr>
</tbody>
</table>

Table 1. UWAgents classes

Each host computer that participates in computation must run at least one UWPlace prior to exchanging UWAgents with other computers. Thereafter, UWAgents navigate over participating hosts autonomously, using the hop() method. The current capabilities of UWAgent include:

1. **Weak migration**: executing from the top of a specific function upon a migration,

2. **Static class transfer**: requiring all agent-carried classes to be declared upon instantiating an agent, and
3. **Asynchronous communication**: allowing a UWAgent to send a mailbox-based message to another agent.

This manual focuses on (1) how to launch a UWPlace at each host computer, (2) how to inject a UWAgent into a collection of UWPlaces, and (3) how to program a UWAgent.

## 2 UWPlace

UWPlace must be launched at each host computer that participates in computation. The following sequence of two commands must be used to launch UWPlace:

```bash
$ rmiregistry port_number &
$ java UWPlace [-p place_name] [-n port_number] &
```

Table 2 summarizes UWPlace’s options and their default values.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p placeName</td>
<td>Place class name</td>
<td>UWPlace</td>
</tr>
<tr>
<td>-n portNumber</td>
<td>Port number</td>
<td>35353</td>
</tr>
</tbody>
</table>

**Table 2. UWPlace command line options**

UWPlace can be launched either from its class file or from the UWAgent jar file, as shown in the following two examples:

**Launching from the UWPlace class file:**

```bash
$ rmiregistry 12345 &
$ java UWPlace -n 12345 &
$ java UWInject localhost TestAgent -n 12345
```

**Launching from the UWAgent jar file:**

```bash
$ rmiregistry 12345 &
$ java -jar $UW_HOME/UWAgent.jar -n 12345 &
$ java -cp $UW_HOME/UWAgent.jar UWInject localhost & TestAgent -n 12345 &
```

1$UW_HOME=/home/uwagent/MA/UWAgent on medusa
2.1 Agent Scheduling

Each UWPlace contains a Scheduler, which the AgentTeamwork Sentinel (or other UWAgents programs) may use to run jobs in a scheduled environment. As part of the scheduling mechanism, the Scheduler provides a resource status reporting function. This allows the Sentinel to make decisions about where to run user processes based on conditions at the current UWPlace (CPU and memory load, number of agents currently residing at the UWPlace, and number of user jobs currently executing).

To use the Scheduler, a UWAgent runs a user process by passing it (as a Thread) to the UWPlace.submit() function. Each UWPlace has a Scheduler associated with it. The Scheduler runs the thread according to a scheduling policy. Currently, only FIFO scheduling is implemented. This scheduling mechanism works as follows: The first thread to be submitted is allowed to run for a given time quantum. Control is then returned to the caller, which may decide to continue running the thread (user process), submit another user process, or migrate to another UWPlace based on conditions at the current UWPlace. If the caller submits another user process, it is placed at the end of the job queue, and does not execute until the first process is finished.

Thread.suspend() is used to suspend a process in order to return control to the caller. If the caller decides to continue running the process, then Thread.resume() is used to resume it. Calling UWPlace.submit(null) instructs the scheduler to run processes that are already in the job queue, without adding any new ones.

For maximum efficiency, the user process should call UWPlace.complete() when it is finished processing. This will preempt the scheduler, and allow the job to end immediately. If this call is not made, the scheduler will not notice that the job has completed until the end of the last time slice.

The following code fragment shows an agent running two user jobs (class ScheduleTestUser) using the UWPlace Scheduler:

```java
UWPlace place = getPlace();
Status status;

// Submit a user job using the scheduler
ScheduleTestUser userJob = new ScheduleTestUser(place);
ScheduleTestUser userJob2 = new ScheduleTestUser(place);

// Submitting job 1
place.submit(userJob);

// Submitting job 2
place.submit(userJob2);

// Get machine status
status = place.getStatus();
DecimalFormat pctFormat = new DecimalFormat("###.##");
String memUsage = pctFormat.format(status.memUsage);
System.out.println("Memory Usage = " + memUsage);
System.out.println("Number of agents = " + status.numAgents);
```
System.out.println("Number of user jobs = " + status.numUserJobs);

// Running job 1 to completion
do {
    status = place.submit(null);
} while (!status.jobCompleted);

// Running job 2 to completion
do {
    status = place.submit(null);
} while (!status.jobCompleted);

3 UWInject

UWInject is used to inject a UWAgent onto a UWPlace. It requires at least two command line arguments. The first argument specifies the IP name of the host where the agent should be injected, and the second argument specifies the file name of the agent to be injected. The optional third and following arguments are passed to the injected UWAgent for its own use. In addition, UWInject accepts the arguments shown in Table 3. These are interpreted by UWInject, and are not passed to the UWAgent.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p place_name</td>
<td>Place Name</td>
<td>UWPlace</td>
</tr>
<tr>
<td>-n port_number</td>
<td>Port Number</td>
<td>35353</td>
</tr>
<tr>
<td>-u path</td>
<td>Path to the file containing the agent to be injected</td>
<td>current directory</td>
</tr>
<tr>
<td>-c client_name</td>
<td>Client user name</td>
<td>Default</td>
</tr>
<tr>
<td>-s sub_class</td>
<td>All sub-classes to be carried with the injected agent. Multiple classes must be delimited by commas.</td>
<td>None</td>
</tr>
<tr>
<td>-m max_children</td>
<td>Specifies the maximum number of children that an agent may spawn. The root agent may spawn one fewer children than other agents.</td>
<td>If not specified, limited only by memory constraints</td>
</tr>
</tbody>
</table>

Table 3. UWInject options

The following examples show how to inject a UWAgent through UWInject:

$ java UWInject localhost TestAgent
   Inject agent TestAgent onto host localhost, without passing any arguments.

$ java UWInject localhost TestAgent 1 2 3
   Inject agent TestAgent onto host localhost and pass arguments 1, 2, and 3.
$ java UWInject mnode1 TestAgent -n 12345
Inject agent TestAgent onto host mnode1 through port 12345, without passing any arguments.

$ java UWInject mnode2 TestAgent 3 4 5 -s TestAgentSub
Inject agent TestAgent onto host mnode2, pass arguments 3, 4, and 5, and instruct the agent to carry TestAgentSub.class with it when a migration occurs.

$ java UWInject mnode1 CommanderAgent arg1 arg2 -s SentinelAgent,BackupAgent
Inject agent CommanderAgent onto host mnode1, pass arguments arg1 and arg2, and instruct the agent to carry SentinelAgent.class and BackupAgent.class when a migration occurs.

$ java UWInject mnode3 TestAgent -u ~/MA/BackupAgent/agents
Inject onto host mnode3 agent TestAgent, which is located in directory ~/MA/BackupAgent/agents.

UWInject was implemented by extending the UWPlace class so that it receives the arguments shown above, instantiates a new UWAgent, and immediately dispatches it to the given UWPlace.

4 UWAgent

4.1 Agent ID

A new UWAgent can be created either from a UNIX shell prompt (using UWInject), or from a UWAgent method called spawnChild(). Although both methods instantiate a new UWAgent, they do not assign the same ID to the new agent. The UWInject approach establishes a new domain, so the instantiated UWAgent becomes a root, and receives “0” as its agent ID. The spawnChild() approach instantiates a child UWAgent in the same domain where the parent UWAgent is working. The child UWAgent receives a new agent ID that is created by concatenating the parent’s ID, a period (.), and a sequential number. The following examples show how agent IDs are assigned to new UWAgents:

- A UWAgent injected by UWInject receives an ID of 0.
- UWAgents spawned by a parent whose ID is 0 receive 0.0, 0.1, 0.2, and so on.
- UWAgents spawned by a parent whose ID is 0.0 receive 0.0.0, 0.0.1, 0.0.2, and so on.

4.2 Agent Programming

To make a Java class runnable as a UWAgent, the user must ensure that the following conditions are satisfied:

- The class must extend UWAgent (“extends UWAgent”).
- The class must implement Serializable (“implements Serializable”).
- The class must implement the void init() method. This method will be called immediately after the constructor.
The class constructors may accept either `String[]` or no arguments.

A method to be called upon a migration (hop) may accept `String[]` or no arguments. Its return value must be of type `void`.

The following code fragment shows the framework for a `UWAgent` class:

```java
class MyAgent extends UWAgent implements Serializable {
    MyAgent() {}
    MyAgent(String[]) {} 
    void Init() {
        hop("nextHost", "function", String [] funcArgs);
    }
    void function(String [] funcArgs) {
    }
}
```

The `hop()` statement is a `UWAgent` method that navigates the `UWAgent` to the destination specified in the first argument, and then calls the function specified in the second argument.

### 4.3 Postponed and Cascading Termination

The UW Agents system implements two behaviors related to agent termination. First, an agent’s termination is automatically postponed until all of its descendants have terminated. The justification for this has to do with how UW Agents communicate with each other. Because UW Agents move around, other agents need some way to keep track of their most recent location in order to send messages to them. This problem is solved by sending messages through the agent hierarchy (e.g., from a child, to a parent, to a grandparent agent). Messages are sent to mailboxes (UW AgentMailbox class). When an agent terminates, its associated mailbox is also terminated. This might cause a communication problem. For example, a parent might terminate, breaking the communication link between a child and its grandparent. To avoid this, an agent’s termination is postponed until all of its descendants have terminated.

In some cases, an agent may want to terminate immediately, without waiting for its descendants to complete on their own. To support this, UW Agents implements cascading termination. To activate this functionality, an agent must call its `setTerminationRequest()` method. Note that cascading termination does not interrupt a user program in the middle of a function, so the parent agent may still have to wait for the current function to complete before its descendants terminate themselves.

### 4.4 Methods

The table below shows the major methods of `UWAgent`. 

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String getAgentId ()</code></td>
<td>Returns the calling agent’s ID.</td>
</tr>
<tr>
<td><code>String getAncestorId (String id)</code></td>
<td>Returns the ID of the calling agent’s parent.</td>
</tr>
<tr>
<td><code>int getChildrenNum ()</code></td>
<td>Returns the number of children that the calling agent has spawned so far.</td>
</tr>
<tr>
<td><code>String getName ()</code></td>
<td>Returns the calling agent’s class name.</td>
</tr>
<tr>
<td><code>String getParentName ()</code></td>
<td>Returns the class name of the calling agent’s parent.</td>
</tr>
<tr>
<td><code>void hop(String hostName, String funcName, String[] funcArgs)</code></td>
<td>Migrates the calling agent to the computer named <code>hostName</code>, and invokes the <code>funcName</code> function there, passing <code>funcArgs</code> as its argument. The argument may be either <code>String[]</code> or null.</td>
</tr>
<tr>
<td><code>void init ()</code></td>
<td>Executed immediately after an agent’s constructor is invoked.</td>
</tr>
<tr>
<td><code>UWMessage retrieveNextMessage ()</code></td>
<td>Retrieves the top message from the received-message queue. If there are no messages in the queue, the calling agent is suspended until a new message arrives. Therefore, the <code>UWAgent</code> should spawn a child thread to receive messages so that the main thread can continue regardless of the message queue status. By calling <code>restartThread</code>, the main thread can instruct the child thread to stop waiting for a new message, and resume its execution.</td>
</tr>
<tr>
<td><code>void restartThread ()</code></td>
<td>Resumes a child thread that is waiting for a new message, regardless of whether or not a message has arrived.</td>
</tr>
<tr>
<td><code>UWAgent spawnChild (String agentName, String[] agentArgs, String DestHost)</code></td>
<td>Loads into memory an agent class from the <code>agentName.class</code> file, passes the <code>agentArgs</code> argument to it, and starts it on the <code>DestHost</code> computer. The <code>agentArgs</code> may be either <code>String[]</code> or null. The instantiated agent starts from its <code>init()</code> method.</td>
</tr>
<tr>
<td><code>UWAgent spawnChild (String agentName, String [] agentArgs, String destHost, String [] classNames)</code></td>
<td>Same as the previous <code>spawnChild</code>, but loads all of the subclasses specified in <code>classNames</code>, as well as the main <code>agentName.class</code> file. Used when the <code>agentName</code> class depends on other classes.</td>
</tr>
<tr>
<td><code>Boolean talk (String recipId, UWMessage message)</code></td>
<td>Sends a message to the agent whose ID is specified in <code>recipId</code>.</td>
</tr>
</tbody>
</table>

Table 4. UWAgent Methods
5 Inter-agent communication

To communicate with another agent, a UWAgent must encapsulate its message into a new UWMes-
sage object, and then pass the object to its talk() method.

5.1 Message-Sending Procedure

The UWMMessage class has the following five constructors:

1. UWMMessage()
2. UWMMessage(UWAgent agent, String msgHeader)
3. UWMMessage(UWAgent agent, String [] msgHeader)
4. UWMMessage(UWAgent agent, String msgHeader, Hashtable msg)
5. UWMMessage(UWAgent agent, String [] msgHeader, Hashtable msg)

The first constructor composes a null message. The remaining constructors create a message consist-
ing of the given String, String array, and/or Hashtable parameters, and associate it with the ID and host
IP name of the specified UWAgent. For example, the last constructor composes a message consisting of
the msgHeader string array and the msg hashtable, and associates it with agent’s ID and the host
IP name where it resides. A hashtable encapsulated in a message must include serializable objects, so
that all the objects are packed in the message and are transferred to a destination UWAgent.

The following example shows how to send a message:

UWMMessage message = new UWMMessage(this, "Hello");
boolean success = talk("0.1", message);

5.2 Message-Receiving Procedure

Inter-agent messages are pushed into their receiver agent’s message queue, which is implemented with
a Vector class. The receiver UWAgent must call retrieveNextMessage() in order to pop and retrieve
the front message from this queue. If there are no messages in the queue, the calling UWAgent will be
suspended until a new message is delivered to it. To prevent itself from being blocked, the UWAgent
should instantiate a new child thread and let this thread call retrieveNextMessage(). The UWAgent can
wake up all of its child threads by calling restartThread() if they are currently waiting for a new message.

5.3 UWMMessage Methods

The user can process a received UWMMessage using the methods shown in the table below.
The following example shows how to receive a message:

// Dequeue a new message from the queue
UWMessage receivedMessage = retrieveNextMessage();
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getSendingAgentId()</td>
<td>Returns the sender agent’s ID.</td>
</tr>
<tr>
<td>InetAddress getSendingIp()</td>
<td>Returns the host IP where the sender agent resides.</td>
</tr>
<tr>
<td>String getAgentName()</td>
<td>Returns the name of the sender agent’s class.</td>
</tr>
<tr>
<td>String[] getMessageHeader()</td>
<td>Extracts a message header into a String[].</td>
</tr>
<tr>
<td>Hashtable getMessage()</td>
<td>Extracts a hashtable.</td>
</tr>
<tr>
<td>String[] getMessageKeys()</td>
<td>Extracts a list of keys from a hashtable (if this message includes one).</td>
</tr>
<tr>
<td>Object getMessageValue(String key)</td>
<td>Retrieves the object corresponding to the given key from a hashtable (if this message includes one).</td>
</tr>
</tbody>
</table>

Table 5. UWMessage Methods

```java
// Retrieve the sender agent’s ID from the message
String senderAgentId = receivedMessage.getSendingAgentId();

// Extract a header from the message into a String[]
String[] header = receivedMessage.getMessageHeader();

// Extract a hashtable from the message
Hashtable table = receivedMessage.getMessage();
```

6 Credits

The following people have worked on UWAgents and the AgentTeamwork system. (Source: http://depts.washington.edu/dslab/AgentTeamwork/index.html)

6.1 Faculty

Munehiro Fukuda  
Computing and Software Systems  
University of Washington, Bothell

6.2 Collaborators

Shinya Kobayashi  
Department of Computer Science  
Ehime University, Japan

Koichi Kashiwagi  
Department of Computer Science  
Ehime University, Japan
6.3 Students

<table>
<thead>
<tr>
<th>Dates</th>
<th>Name</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/03 - 06/03</td>
<td>Hyon Kim</td>
<td>Designed UWAgents’ agent migration.</td>
</tr>
<tr>
<td>04/03 - 09/03</td>
<td>Eric Nelson</td>
<td>Ported Tsukuba Univ’s Voyager-based job dispatcher to LAB302 and designed UWAgents’ inter-agent communication.</td>
</tr>
<tr>
<td>04/03 - 06/03</td>
<td>John Hagen</td>
<td>Designed a computing-resource database with MySql.</td>
</tr>
<tr>
<td>06/03 - 09/03</td>
<td>Doug Kim</td>
<td>Designed the Sentinel agent’s job launching feature.</td>
</tr>
<tr>
<td>06/03 - 03/04</td>
<td>Ryan Liu</td>
<td>Designed a Xindice-based resource database and the Resource agent.</td>
</tr>
<tr>
<td>04/04 - 06/04</td>
<td>Vivian Chan</td>
<td>Ported the Java Grande benchmark to AgentTeamwork.</td>
</tr>
<tr>
<td>04/04 - 06/04</td>
<td>Tae Suzuki</td>
<td>Re-engineered and documented all mobile agent code.</td>
</tr>
<tr>
<td>06/04 - 09/04</td>
<td>Duncan Smith</td>
<td>Implemented and enhanced UWAgents’ priority-based scheduling and inter-agent communication.</td>
</tr>
<tr>
<td>06/04 - 09/04</td>
<td>Donya Shirzad</td>
<td>Ported MPI applications to AgentTeamwork.</td>
</tr>
<tr>
<td>06/04 - 09/04</td>
<td>Shane Rai</td>
<td>Enhanced the Resource agent.</td>
</tr>
</tbody>
</table>

7 Final Comments

The UWAgents mobile agent platform was originally developed for our mobile-agent-based PC grid, AgentTeamwork, and it is being used for that purpose. However, we welcome comments from users who are interested in using this platform for their distributed/network computing applications. Any comments and/or suggestions would be greatly appreciated. Please contact us at:

<table>
<thead>
<tr>
<th>Name</th>
<th>E-mail Address</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Munehiro Fukuda</td>
<td><a href="mailto:mfukuda@u.washington.edu">mfukuda@u.washington.edu</a></td>
<td>For English/Japanese assistance</td>
</tr>
<tr>
<td>Koichi Kashiwagi</td>
<td><a href="mailto:kashiwagi@koblab.cs.ehime-u.ac.jp">kashiwagi@koblab.cs.ehime-u.ac.jp</a></td>
<td>For Japanese assistance</td>
</tr>
</tbody>
</table>