LuaMAP

Andrew Arnold

# System Synopsis

LuaMAP stands for Lua Mobile Agent Platform. The system allows Java applications to transmit, execute, and expose resources to Lua-based Mobile Agents. The project will consist of several applications, libraries, and Agent scripts:

* A Java library which implements the Mobile Agent Platform as a P2P system, which requires open ports.
* A Java Library which implements it as a client-server system, which does *not* require open ports, making it suitable for personal computers or any other machine which is behind a firewall or machines behind NAT.
* A containerized application which implements the Client Broker service, as detailed later.
* Lua scripts used by the two libraries, such as Script Libraries and Agent Facility auxiliary scripts.
* A Lua script which implements an Administration Agent, as detailed later.
* A Lua script which implements an ADNS Agent, as detailed later.

Network communications are conducted using HTTP, and the server version of the platform acts as a REST API. Any given platform instance can support any number of concurrently executing agents, and these agents can communicate with each other locally or over the network via HTTP. The agents have limited access to local files and can “carry” files with them from platform to platform.

A LuaMAP system may be viewed as a set of RESTful APIs, which internally use Mobile Agents as their basic unit of computation. This is due to the fact that any and all communications in this system are conducted using HTTP and follow standard REST conventions. Client-Mode Platform instances are an exception to this, as they behave as a client which merely *accesses* the set of APIs.

# Component Synopses

## Agent Lua Environment

### Exposed Facilities

#### Exposing Application Access via AgentFacility.class

Application resources may be exposed to agents by extending the AgentFacility class. This class contains logic for sandboxing child classes in the agent’s Lua environment.

The child class can directly expose one of its methods as a Lua method by annotating it with the @ExposedMethod annotation, and optionally providing a new method name. Exposed Methods aren’t directly invoked by the agent’s Lua code, but rather they are sandboxed within a Lua table containing a proxy function.

The child class can also generate its own Lua objects to place in the sandboxed Lua table by annotating a public, String-returning, no-parameters method with the @SyntheticLuaObject annotation and providing the table index to store the object under. The returned Java string must be valid Lua source code, which when executed will return the object to store within the sandboxed Lua table. Within this source code, there are two Lua tables only accessible by synthetic lua object source code. These tables are “facility” and “proxy”.

The “facility” table is the Java object which has been placed directly into the Lua engine and can be used to directly invoke all public methods within the child class. It is recommended to only access methods from this table, and the Lua method syntax *must* be used to invoke them.

The “proxy” is the sandboxed proxy table which will be accessible by agents. At the time of execution of this source code, the table is fully accessible, but is not so by the time agents will execute. Direct access to this table is useful for programmatic generation of its members.

Within synthetic Lua object source code, if access to other agent facilities is desired, it is *very important* to note the load order of the facilities. It is recommended to test the existence of the facility proxy tables by accessing \_G[“facility\_name”], and then accessing table members from the result.

#### Default AgentFacility classes

##### Platform

Platform: This facility allows for agents to interact with the platform directly. These allowances include jumping to another host and various configuration features.

##### Serial

Serialization: This facility provides (de)serialization functions for Lua objects. This uses the same mechanisms used for agent transmission and handles most of the ceremony required for Eris serialization. It also deals with some of the string limitations of the Lua engine.

##### IAC

Inter-Agent Communications: This facility encompasses all the Inter-Agent Communications features of the platform. It contains functions for simple and intuitive messaging and channel management between agents both locally and remote.

##### MAData

Mobile Agent Data: This facility provides functions for registering and manipulating agent data files.

##### LIO

Local I/O: This facility provides limited functions for accessing the host’s local file system. These functions may be restricted by the platform and the restrictions may vary from agent to agent.

##### Logging

Logging: This is an adapter for TinyLog2. Agents by default carry an agent data file called “log” which is written to using some default TinyLog2 features.

##### Shell

Shell: This is a facility for interacting with the host’s shell. As with LIO, this facility may have restrictions and they may vary from agent to agent.

##### Security

Security: This facility provides various security features like encryption, hashing, and secure PRNG.

##### ADNS

Agent Domain Name System: This facility provides functions regarding the ADNS. Agents can request address translation for other agents and can (un)register themselves with ADNS Agents.

### Script Libraries

Script libraries are custom Lua libraries available to agents. These libraries are sandboxed so that the Agent Lua Environment can rely on them. Some of these include advanced table functions, sandboxing utilities, data structures, algorithms, and other useful Lua functions. Scripts Libraries have one requirement, and it’s that they be pure Lua code. The libraries are loaded by the Platform during the creation of a new Agent Lua Environment, rather than being dynamically loaded like traditional Lua libraries.

### Lua Interpreter

The Lua Interpreter used by this library is a fork of JNLua owned by the Terasology dev team. This fork incorporates support for the Eris library, which is a powerful, heavy-duty serialization library maintained by Sangar. It retains most of the performance of normal JNLua; however, because agents aren’t compiled, the performance of this system is not as good as it could be if they were compiled.

### Special Limitations

The conversion between JVM Strings and Lua strings is, unfortunately, far from perfect. conversion of certain ranges of characters causes a generic engine crash which is nearly impossible to debug, and the work-around is to temporarily store the ASCII codes of individual characters into a Java data structure, place that object into the Lua engine, and then build a new Lua string by translating those codes back to characters. This drastically reduces the performance of string operations in certain situations, and because of this is recommended to avoid passing many strings between the JVM and Lua Engine instances.

## Agent Execution

Agent execution is implemented by the deserialization or interpretation of the agent’s serialized form or source scripts, respectively, being placed in its own new Agent Lua Environment, and being submitted to the host’s execution pool. The execution order of the agents will be determined by the inner workings of Java’s thread pool libraries. Once an agent has yielded (via Lua yield()) to the environment’s main process, the platform will determine the next step for this agent.

If the agent requested a jump via the Platform agent facility, the platform will serialize the agent, perform any ADNS protocols if requested, and then perform the Agent Transmission Protocol with the host targeted by the jump.

If the agent did not request a jump, the platform will terminate the agent and erase its agent data. Afterward, the platform will notify the host that the agent originated from if that information was provided by the agent’s metadata. This will be logged for debugging purposes.

## Agent Data

Agents may have data that is carried with them across jumps, and this data may be accessed at any time by the agent. It is deleted from the host platform as soon as an Agent successfully jumps or terminates. Each Agent executing in a Platform instance has a directory associated with it, and its Agent Data is stored as a file with a random UUID file name. These random file names are mapped to the Data’s identifier internally by the platform. Access restrictions are much more relaxed for Agent Data than for a host’s local files, making them suitable for an Agent’s light-weight data dependencies, such as child scripts or configuration files.

In the far future of this project, this data could be dynamically stored in memory and/or disk, to speed up access for short-lived agent executions and/or sufficiently small data. This would require sophisticated memory management comparable to that of an OS, which will require a very large sum of development.

## Client Broker for Client-Mode Platforms

Platforms may operate in “Client-Mode”, which is intended for use on hosts that are behind a firewall or are otherwise unable to reliably listen to the standard LuaMAP port. This mode is less efficient, and drastically increases the latency of Agent jumps and IAC messaging. To make this mode possible, there must exist some remote service which maintains “inboxes” for these restricted Platform instances. This service is the Client Broker.

The Client Broker behaves similarly to the surface-level behavior of an email server. Client-Mode Platform instances (hereby *Clients*) may register themselves with this service and be assigned a random UUID. After a Client has registered, it may make API queries to this service to see the items in its inbox. From the perspective of all Platform instances which wish to communicate with a Client, the Client’s Platform Root URL will consist of the address of the Client Broker and the random UUID of the Client.

## Agent Reception Services

### Server Implementation

When an API query has been made to the reception services of the platform, the service will perform the receiving role of the Agent Transmission Protocol. Once this has been completed, the agent will be processed by the execution service.

### Client Implementation

This implementation receives incoming agents by polling a Client Broker for incoming Agents, and then conducting the transmission protocol (as described in the Server Implementation) with the Broker.

## Agent Transmission Service

When an agent jumps between two hosts, it is transmitted via the Agent Transmission Protocol. The protocol is as follows:

An agent calls Platform.jump() and passes the API root of the desired host to the function, as well as any ADNS data. Afterward, it calls coroutine.yield() to yield its execution to the environment process. The environment is retained until the end of the protocol. The platform then makes an API query using the agent-provided host platform API root. This is a HTTP POST query, and the data provided includes the agent’s metadata, the file sizes of its agent data, and the API root of the agent’s origin host (if available).

When the receiving host receives this request, it may reject if for any reason. If it does not, it will send a positive acceptance response and begin allocation of a new Agent Lua Environment and files for the agent data.

If the request results in anything other than a positive acceptance response, the Transmission Failure state of this protocol is entered. Otherwise, the requesting platform performs API queries for each file of agent data and transmits the file contents over the HTTPS stream. If any one of these queries results in a negative (or no) response from the receiving platform, the Transmission Failure state is entered. After transmitting the agent data, the agent serialization is transmitted via another API query. If a negative (or no) response is received, the Transmission Failure state is entered.

After successfully transmitting the agent, the requesting platform will delete the agent data files, perform any ADNS service requested, and then remove the agent’s process from the execution pool.

If any part of the transmission process enters the Transmission Failure state, the requesting platform will cease the transmission process and resume the agent process where it left off. The agent will resume execution as if it had successfully transmitted. The agent can discern between a successful jump and a failed jump by polling the API root of the current host before and after the jump attempt.

## Inter-Agent Communications Services

### Intra-Platform Communications (local)

Agents executing on the same host may communicate directly via a basic message-passing system maintained by the host platform. These messages still use the IAC format, to maintain consistency. These messages are deposited into the IAC message queue for the receiving agent.

### Inter-Platform Communications (remote)

Agents executing on different hosts may communicate by sending messages to one another. This is done by targeting the agent’s ID via an API query to the receiving agent’s current host. Information about the host may be obtained through an ADNS or by some other scheme implemented by the agents.

## Agent Domain Name Service

An Agent Domain Name Service (ADNS) acts similarly to how Domain Name Services do within the Internet. An ADNS maintains a table which maps the Agent UUID of any registered agent to its reserved Domain and the IAC Root URL of its most recent host Platform instance. This service is implemented as a normal Lua Mobile Agent in the MAP system, which will *not* jump after it has started its service routine. The ADNS Agent receives updates via its host’s IAC Service.

Platform instances which support ADNS participation will facilitate support via their IAC Service. Whenever a participating Agent requests transmission, it will send an additional argument to *jump()*, which should be an ordered list of preferred ADNS Agent IAC URLs; If the host Platform supports ADNS participation, the host Platform will send an update to the requesting agent’s preferred ADNS Agent via the IAC Service.

Agents may request ADNS-registered Agents’ UUID and most recent Platform IAC Root URL by sending an IAC message to an ADNS Agent containing the Domain Name of the requested Agent. The ADNS Agent will reply by sending an IAC message to the requesting Agent containing the requested Agent’s UUID and its most recent Platform instance’s IAC Root URL.