The IMPACT Agent instantiation life-cycle

(from conception to run-time)

2.0 An IMPACT Agent instantiation life-cycle can be broken down into three phases, development, deployment, and run-time as follows:

2.1: Agent development (a brief recap) -- The IMPACT agent development environment (AgentDE) comprises a set of compilers which translates an input agent definition (text) into an instantiated hierarchy of IMPACT agent core classes (*Java* objects) which capture the desired agent behavior. The `IAD_MetaData` class, a *Java* thread class extension, heads the hierarchy; It provides the agent control interface and ensures that every agent can run independently in its own thread.

All IMPACT agent core classes implement the standard *Java* "serializable" interface; This allows us to save/restore class objects to/from disk and transfer them to remote clients as necessary via standard *Java* Remote Method Invocation (RMI) library calls. Thus, when a developer prompts the AgentDE to save an agent instantiation, the system serializes the agent core class hierarchy into a system independent binary file we call the agent meta data (AMD) file. When restoring an agent from its AMD file, the IMPACT deserializes *Java* class objects from the AMD binary file, reinstantiating the agent core class hierarchy.

2.2: Agent deployment -- Deployment to target roost(s) is handled via one of the following three methods:

2.2.1: Physical deployment – The developer physically copies the AMD serialization files to the target roost initialization directory and then prompts the roost to load from the directory at start-up. In this mode, the roost server builds an agent list from the AMD files found in the initialization directory (at start-up), registers itself and the loaded agents with the yellow-pages server (`IYR_YPHook.registerAgents` *Java* RMI method call), and then enters the message queue / duty watch processing loop.

2.2.2: Developer prompted deployment via roost client/server calls (PENDING) **–** At start-up, the AgentDE obtains a listing of active roost servers from the yellow-pages server. Once the developer is satisfied with the test results for a given agent, they can select the “Deploy program” button to prompt the agent deployment dialog-box. Here they select the desired target roost(s), verify the run-time scheduling, and initiate the code transfer via the “Deploy” button. At this point, the AgentDE system contacts the target roost server(s) and begins the agent class object transfer through the network via the `IRR_RoostHook.inProcessAgent` *Java* RMI method call (see Image 2.1). Once the code transfer is complete, the target roost server serializes the agent class hierarchy to a local AMD file, adds the new agent run-time data to its list, informs the yellow-pages server of the new addition (`IYR_YPHook.registerAgent` *Java* RMI method call), and continues its message queue / duty watch processing loop.

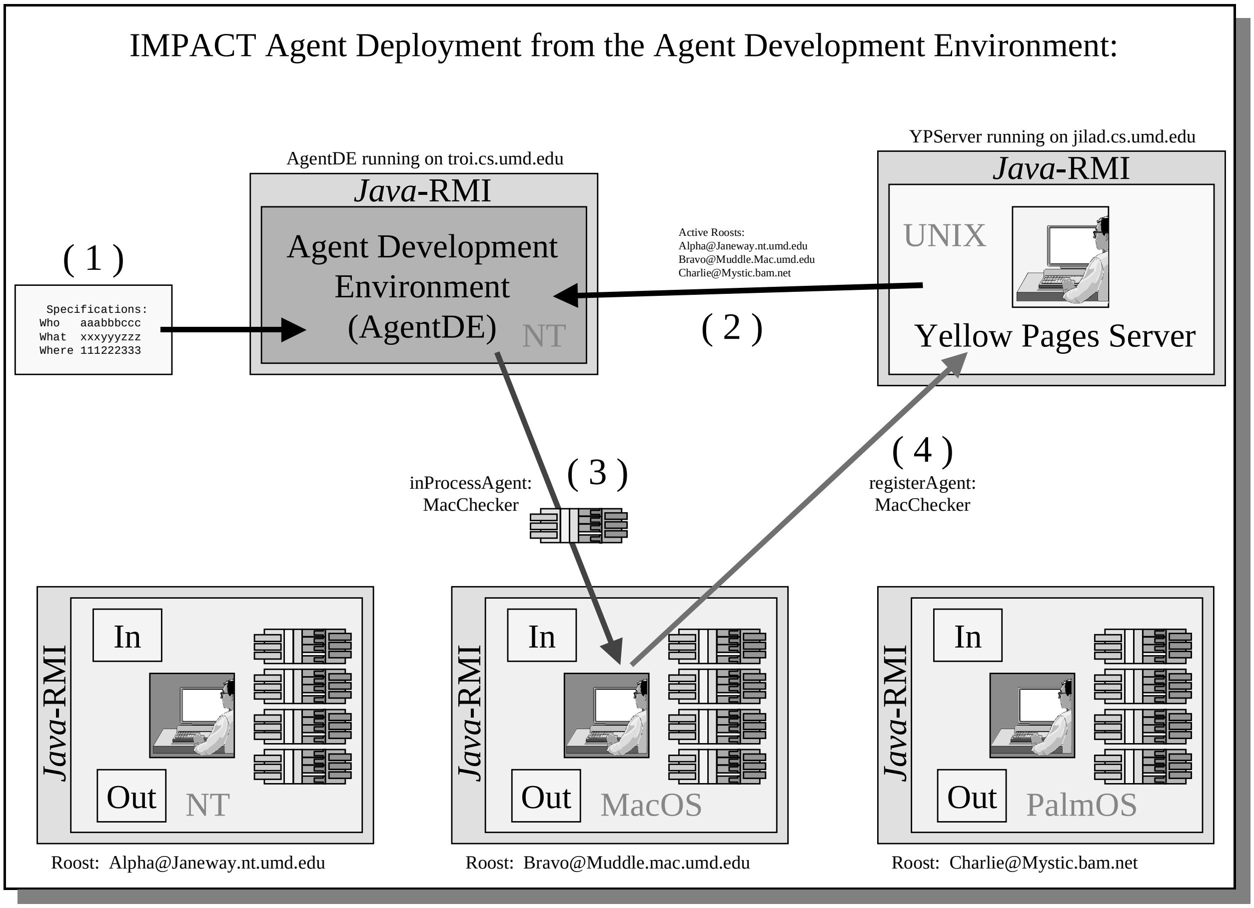

Image 2.1: Developer prompted agent deployment.

2.2.3: Agent prompted deployment via roost client/server calls (PENDING) **–** An agents behavior specification might include the necessity that it physically moves from roost to roost, while processing and/or generating data; this we call a mobile IMPACT agent. In this case, the agent instantiation encodes the constraints and parameters which, during its normal execution cycle, prompts the current roost host to temporarily suspend the mobile agents execution thread, contact the next roost host, and send the agent on its way through the network via the `IRR_RoostHook.runTDY_Agent` *Java* RMI method call. In this case the new roost host does not register the wayward agent with the yellow-pages server. Rather, the new roost host simply continues the mobile agents execution thread from where it stopped; The agent continues as it was before, except that it now has access to assets available from the new roost host.

2.3: Agent run-time – An IMPACT roost server provides an environment in which agents can work, communicate, and sleep. Since it's written in *Java*, roost servers can exist on any platform for which the *Java* runtime library is available (currently version 1.2). This forms the framework through which the IMPACT agents can operate collaboratively while widely distributed across networked resources, regardless of platform and/or operating system. Within a roost server is the "duty-officer" (`IR_DutyOfficer`) class object which functions mainly as a communications and activity daemon for agents deployed to its server.

2.3.1 Local message processing: As messages arrive for local agents, the duty officer "wakes" the agents by deserializing the appropriate AMD file (noted in it's agent run-time list) into the target agent's *Java* class instantiation, and then starting the thread execution. Once running, the agent follows its specification to completion.

2.3.2 Message generation: Agents communicate with other agents by generating and inserting messages (`IA_Message` *Java* class objects) into the message queue (`IA_MessageQueue` class object). As running agents generate messages for other local and/or remote agents, the message queue, in-turn, consults the yellow-pages server as necessary to obtain the actual message routing data (this is accomplished through the network via the yellow-pages client `IRR_YPHook.getRoostings` *Java* RMI method call).

2.3.3 Remote message processing: As with the IMPACT agent core classes, the `IA_Message` class implements the *Java* serializable interface; Thus message transfer can be conducted between roosts via standard *Java* RMI library calls. For every "out-going" message (those destined for other roost servers) in the queue, the duty-officer contacts the target remote roost server and transfers the message object through the network via the `IRR_RoostHook.addMessage` *Java* RMI method call. Through this procedure, the remote message objects are removed from the local message queue, and transferred to the remote message queue for consumption by the remote agent.

2.3.4 Run-time scheduling: Part of an agents behavior specification can be that of a run-time schedule. That is to say that agents can be designed to run at regular intervals, or some designated specific, perhaps recurring, schedule. Thus the duty officer must periodically review the run-time data and wake those agents whose schedule constraints match the current date-time group (DTG); As before, the duty officer wakes a given agent by deserializing from the appropriate AMD file to instantiate the necessary IMPACT agent core class hierarchy, and then starting thread execution.