Ubiquitous Computing
University of Illinois at Urbana-Champaign
Computer Science Department
Request for Comments: 0001
Obsoletes: none

Christopher K. Hess
Renato Cerqueira
Manuel Roman
Software Research Group
September 2001

The Gaia Bootstrap Protocol (GBP)

Table of Contents

The Gaia Bootstrap Protocol (GBP) 1
1 Status of this Memo 1
2 Introduction 1
3 Requirements 2
4 Services 2
5 Protocol 3
5.1 Interface Repository 4
5.2 UOBHost 4
5.3 Name Service 4
5.4 Other Services 6
6 Mobile Devices 6
7 Inter-Space Discovery 6
8 Known Bugs 6
9 Appendix 6
9.1 Example `services.lua` File 6
9.2 Example `as_cfg.lua` File 7

1 Status of this Memo

This document is the design specification for the bootstrapping protocol for Gaia, an implementation of an active space. Any comments, changes, or suggestions are encouraged.

2 Introduction

Active spaces (AS) are the infrastructure to coordinate the hardware and software in a physical space ("space"). The collective hardware in the space is treated as a large distributed computer and the active space kernel is responsible for managing the resources and devices in the space, similar to the way in which an operating system manages the resources on a single computer. Before applications may execute in the space, the machines hosting the infrastructure must be running and the core AS kernel must be started. In Gaia, the kernel is composed of several core services that are required by all applications. These include the naming service, file system, event service, discovery service, and space repository, etc. The goal of the Gaia Bootstrap Protocol (GBP) is to automate the steps needed to start the kernel services required by Gaia applications. The AS creates a virtual space area network on top of the underlying local area network (LAN).

3 Requirements

Active spaces should be able to function autonomously from other spaces; that is, an AS should not depend on any outside services to be available for correct operation of the local space.[1] For example, an AS should be able to run on a computer that is totally disconnected from the network. In addition, a physical space, such as an office, may be split into several ASs. Even if they share the same underlying network, they should be able to operate independently from one another. In addition, there must be a scalable mechanism to discover services and entities within a space. These requirements adds some minor complexities to several parts of the system, such as service discovery, since they are often found through multicast, and the system must take care to avoid service discovery collisions.

Based on the above-described requirements, the infrastructure must lie on a multicast or broadcast network to facilitate local discovery. For standalone machines, this may be achieved through the loopback interface. Consider the case where two adjacent rooms are on the same LAN. In this scenario, it is not possible to use multicasting without possible conflicts. A multicast message sent on the LAN to discover a service may result in two responses, the order of which can differ on subsequent queries. Therefore, in order to isolate the interference of kernel services between spaces, there must be a policy in place to prevent such interference from occurring.

If more than one network is present in a space, they must be cross-reachable via multi/broadcast. If this were not the case, there may be multiple instances of the directory service (described below) running within

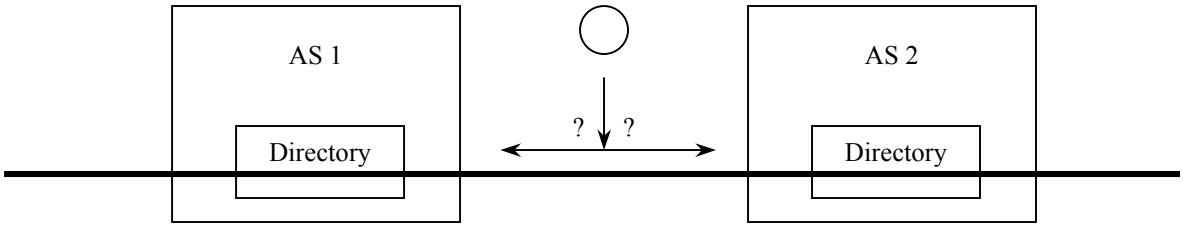

Figure 1 Problems can occur when two active spaces reside on the same LAN.

a space, which would not be synchronized. Certain services that were registered in a certain instance of the directory might not be found, even though they may be running in the space. Therefore, a further requirement is that there be a single instance of all kernel services running in a space. This is implicitly derived from the requirement that ASs be autonomous.

4 Services

The Gaia kernel is composed of several essential services. The order in which they are started is important, since there are certain dependencies among them. Components communicate with the kernel services through remote method calls. Before a service can be used, it must be discovered and a reference to it must be obtained through some directory service. The directory service used in Gaia is the CORBA Naming Service (NS). The NS acts as a central repository where kernel services can be registered and discovered. Services are registered within a space with well-known names under the root naming context, allowing them to be discovered in an identical manner in each space. Once an object reference is obtained, communication continues directly between the client and the service component. Although the NS may

[1] The introduction of security in the system will change this slightly. Security policies will be administered per domain, or related collection of spaces. A security service will have to be operational to authenticate components and secure communication channels. However, the security services are separate from the bootstrap process, which applies to a single space, rather than a domain. Once the security services for a domain have been started, ASs can be bootstrapped independently.

appear to be a bottleneck in the system, this is typically not the case, since in Gaia, there exists one NS per space (rather than having one NS per LAN), and the NS is only contacted to retrieve or register an object reference.

Services are broken up into two types; *space services* and *host services*. There exists a unique instance of each space service per physical space. Host services run on a specific machine in a space and there may be more than one instance of such a service. For example, a device or sensor may be attached to a particular machine and the service that exports it must be run on the same machine to which it is connected. The following table enumerates the space and host services that are started as part of the bootstrap process, including the order, dependencies, and short descriptions.

Order	Service	Dependencies	Type	Description
1	Interface Repository	None	Space	Contains the interfaces to the rest of the services. Used by the rest of the bootstrap procedure to start and communicate with the remaining services.
2	UOBHost	Interface Repository	Host	Allows containers and components to be started on a particular machine.
3	Naming Service	UOBHost	Space	The central registry for the remaining services. Services are registered in will-known location so that they may be found to components that wish to use them.
4	Event Manager	Naming Service	Space	Allows decoupled communication among services and components.
5	Presence	Event Manager	Space	Detects the presence of entities on the presence channel.
6	Informer	Event Manager	Space	Detects the presence of people on the presence channel.
7	Trader	Naming Service	Space	A database of entities in the system. Can be queried for entities or components satisfying particular constraints.
8	Space Repository	Event Manager Trader	Space	Front-end to the trader that provides persistence through an XML database.
9	Layout Manager	Naming Service	Space	Describes the data layout of a space, including the personal data of users.
10	Container Manager	Layout Manager	Host	Exports storage associated with a particular machine.
11	Entities	Space Repository	Host	Devices and sensors that are physically connected (often through a serial connection) to a particular machine.

Table 1 Kernel services that are started as part of the bootstrap process must be started in a particular order to ensure that the dependencies among services is preserved.

5 Protocol

For proper operation of the kernel, there must be a unique instance of each global service running in a space. The GBP ensures this property by probing for existing instances of a service before starting them.

A space can be configured depending on the available machines and the desired layout of the machines in the space. For example, all services can run on one machine, or may be distributed on different machines within the space. The bootstrap process does not complete until all services are running, or an error is detected. The GBP uses the Lua scripting language and the LuaORB to start the services. The bootstrap scripts are split into five parts:

1. `boot.lua` – includes the main driver script to run the protocol
2. `bootstrap.lua` – includes the functions to perform the bootstrap process.
3. `services.lua` – list and configuration of services to be started.
4. `as_cfg.lua` – includes the space specific configuration.
5. `local_cfg.lua` – includes machine specific local configurations.

Each space is assigned a unique space id (SID), which could be a room number in a building, airport code in a flight terminal, etc. The same set of scripts should be accessible to each machine that is to be a part of the bootstrap process. The `as_cfg.lua` configuration script contains the SID and `services.lua` contains a list of the services that are to be started and any special configuration information that is required for each. The configuration for each service specifies the machines that the service is allowed to run. This list is used to determine if the current machine should attempt to start a service if it is not already running in the system.

The bootstrap process iterates over the list of services that are to be started. The protocol first probes the network to see if the service is already running. Depending on the service, this is done in different ways, and is described below. If the service is determined to be running already, the protocol moves to the next service in the list. If the probe of the service is unsuccessful, the protocol checks to see whether the current machine has been designated to run the service. If so, the service is started on the local machine, otherwise the protocol waits for the service to be started by another machine in the network.

TODO: more detail of scripts

5.1 Interface Repository

The first service that must be started is the Interface Repository (IR). The LuaORB requires the interfaces of the services it needs to communicate with to be available in order to perform type checking. Since the IR is started before the NS, it cannot be registered in the NS. Therefore, it is started on a known host and port.

TODO: describe how IR is bootstrapped

5.2 UOBHost

After the IR is started, the UOBHost is started on the local machine and is used to start all subsequent services. The UOBHost is used to start local and remote components and containers for components. Since the UOBHost is started before the NS, it must have no dependency on the NS. Therefore, it is bound to a well-known port on the local machine. In order to start a component on a particular (possibly local) machine, a CORBA request is sent using the `iioploc` mechanism, therefore bypassing the NS to contact a UOBHost. This does not violate the location transparency exhibited in the other services since the machine name must be known in order to start a component on it.

5.3 Name Service

Since the NS is the central database for storing object reference, it must be started before references can be entered into it. This is typically achieved through multicast, where the NS listens on a well know multicast group address and port. This approach does not work in the AS environment, as explained above, since

there may be multiple NSs per LAN. In order to avoid collisions, each NS must respond only to queries for the space that it is supporting.

Gaia was designed to use off-the-shelf services. For example, Gaia uses standard implementations of the CORBA name, trader, and event services. However, many implementations use multicast for discovery and therefore are restricted to one service per LAN. In order to circumvent this limitation, Gaia employs a Name Agent (NA) that is used to filter requests that only responds to those requests originating within the space it is supporting. When the NS is first started, it writes its reference IOR to a file. The NA is configured to respond only to requests for a given SID, which is placed in each request. The NA is run on the same machine and waits for this file to become available and reads the IOR into memory. The NA then listens on a well-known multicast port and address, receives all requests, and filters them based on the SID it was configured with, and responds with the IOR if the SID in the request matches its own.

The steps taken to resolve the NS are as follows:

1. The NS is started via the UOBHost in a space if no other instance is determined to be running. The IOR of the root naming context is written to persistent storage.
2. The NA is started and listens on a different port than the NS. The NS port is not used, but is made different to eliminate port allocation collisions. The NA reads the IOR from persistent storage. The NA is configured to filter out requests not directed for it. The NA will periodically look for the existence of the IOR. If it is not available, perhaps because the NS has not started yet, it will periodically poll until it is found. Once the IOR is found, it is cached in memory and the NA continues by joining a multicast group and binding to a certain port.
3. A component sends out a request to discover the NS. It waits for a limited amount of time for a response.
4. The multicast request is send on the LAN. All NAs tuned in to the multicast group receive the request for the NS IOR.
5. Each NA filters the request based on the SID that is encapsulated within the request. The NA that has been configured with the same SID accepts the request.
6. The IOR of the NS is unicast back to the requester in a UDP packet. If the requester does not receive the response within a specific timeout period, it attempts resolve the NS two more times. A response may not arrive either because a packet was lost or there is no NA for the space. If, after three attempts to contact the NA and no response is returned, the NA/NS is assumed not to be running. Once the NS is running, responses return with no delay.

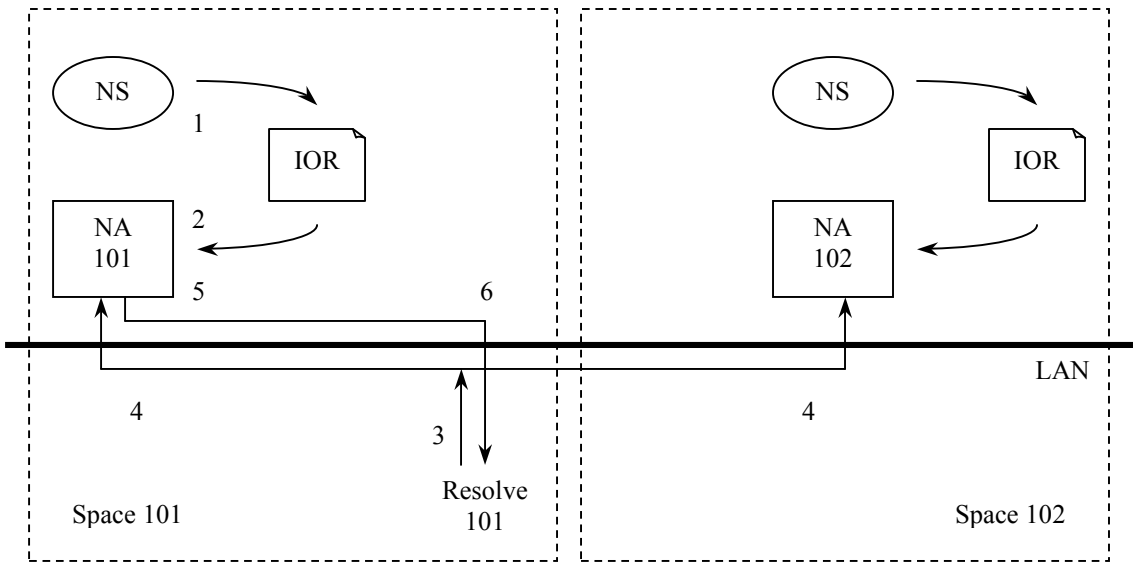

Figure 2 The Name Agent is used to filter requests based on the space identifier. This allows multiple instances of a Name Server to be running on a single LAN.

5.4 Other Services

All services started after the NS are registered in the NS. The probing mechanism used to discover these services simply queries the NS to see if they are registered there. Each service registers itself in the NS. When a service registers itself, it discovers the NS by using the mechanism described above, using multicast and contacting the NA for the space it is in. Therefore, if a service reference is found in the NS, it is assumed to be running in the space. The potential for race conditions is eliminated through the object binding mechanism. If an object tries to bind itself and an object of the same type is already bound, the binding will fail. Therefore, the first received bind request will succeed; failure to bind will signify that another process has bound itself in between the time that the probe failed and the bind was issued.

6 Mobile Devices

Mobile devices must be treated differently from stationary machines. Mobile devices typically communicate through wireless links, such as IEEE 802.11. However, it may not be possible to multicast from the wireless network to the network underlying the space, due to administrative domains or other technical hurdles. In such situations, the device must use an alternate method for obtaining a handle on a space; one such method is through the use of infrared beacons. Once the NS has been activated in a space, a service is started that periodically sends the IOR for the space out into the space via infrared. Physically local devices can then pick up the signal and use the IOR to obtain any of the other services or components that are available in the space and can henceforth register themselves in the NS is required.

7 Inter-Space Discovery

In order for ASs to interact, they must get a reference to the NS corresponding to the space with which it wants to communicate. If the ASs reside on the same multicast network, the SID can be specified when resolving the NS for the space. From that point, any service or component in the space can be obtained. If the ASs cannot communicate via multicast, the NS reference must be obtained through some out-of-band mechanism, such as a directory service similar to DNS.

8 Known Bugs

A race condition exists when two machines that are configured to start the NS are started at the same time. It is possible that both machines will probe for the existence of the NS and both determine that it is not running. In such a scenario, it is possible that more than one NS will be started in the same space. A random backoff mechanism may be used to probabilistically solve this problem. This must be investigated further in future releases of the NA.

The short-term fix is to configure the space with exactly one machine designated to run the NS. All other machines in the space will wait for the designated machine to start the NS before they may continue.

9 Appendix

The following sections describe some example configuration files.

9.1 Example `services.lua` File

A partial listing of a `services.lua` file is given below. The file illustrates some of the service-specific options that are available. Note the `hosts` configuration parameter; it is used to specify the host machines that the particular service is allowed to run on. To run an AS on one machine, the `host` parameter should be configured to be the local machine. Also note the UOBHost kernel service; it has been configured with

host_policy of all, signifying that is should run on every machine that the script is executed, i.e., it should run on every machine in a space willing to run services and components.

```
KernelService{
   name = "InterfaceRepository",
   interface = "CORBA::Repository",
   hosts = {"host1"},
   port = 7777,
}

KernelService{
   name = "UOBHost",
   interface = "Gaia::Entities::Devices::UOBHost",
   host_policy = "all",
   port = 8888,
}

KernelService{
   name  = "Naming",
   interface = "CosNaming::NamingContext",
   hosts = {"host1","host3","host3"},
   port = 8765,
}

KernelService{
   name = "EventManager",
   interface = "Gaia::Events::EventManager",
   hosts = {"host3","host4","host5"},
   event_types =
{"DEVICE","SERVICE","PERSON","APPMODELUPDATES","SYSTEM"},
   event_channels = {
      {name="DISCOVERY",type="PERSON"},
      {name="PRESENCE",type="PERSON"},
      {name="DISCOVERY",type="DEVICE"},
      {name="PRESENCE",type="DEVICE"},
      {name="DISCOVERY",type="SERVICE"},
      {name="PRESENCE",type="SERVICE"},
      {name="COMPONENTCREATION",type="SYSTEM"},
      {name="COMPONENTDESTRUCTION",type="SYSTEM"},
      {name="ERROR",type="SYSTEM"}
   },
}
```

9.2 Example as_cfg.lua File

The as_cfg.lua file is used to configure timeout values and the space identifier.

```
ASConfig{
  AS_ID = "2401",

  timeout = {
     DefaultSleep = 1,
     WaitKernelService = 60,
     WaitNaming = 60,
  },
}
```