Discovery Service

Madhavarapu Jnana Pradeep

{madhavar@cs.uiuc.edu}

1. Overview

Distributed systems are very dynamic and keep changing ceaselessly. Various components are loaded and unloaded as the necessities keep changing. The dynamism of such systems is further increased by component crashes, which are inevitable in any distributed system. In such systems, it is necessary to keep track of components currently active. In Gaia, this is achieved by the Discovery service. The Discovery service tracks software components as well as persons present in an ActiveSpace. Information about currently active components is required for diagnostics and 'self-healing' activities that stabilize the system by handling component failures. Information about persons present in an ActiveSpace is required for setting user preferences and many other functions. This document discusses the implementation of Discovery service in Gaia.

2. Introduction

The Discovery service is a service in Gaia that keeps track of entities present in an ActiveSpace at any time. These entities can be services, devices and even persons. Presence and Informer services are the two major components of Discovery service.

An ActiveSpace can have a lot of services and devices running in it at the same time. Component failures are inevitable in any system and have to be tracked. For this reason, the Presence service keeps track of which entities are currently running in an activespace. All service and device entities send heartbeats at regular intervals, to show that they are up and running. The Presence service listens to and filters these heartbeats to know which entities are currently running in the activespace. Whenever a new entity is discovered, Presence sends a message announcing the starting of the new entity. When an existing entity stops sending heartbeats, Presence service sends a message announcing that the entity has stopped. Thus, component crashes can be detected and action can be taken to heal the system.

While Presence service keeps track of service and device entities in an activespace, Informer service keeps track of the persons present in an activespace. Persons in an activespace are detected by devices such as RF badge detectors, cameras etc. All such sensing devices send periodic updates of which persons have been detected in the room. The Informer service listens to and filters these information packets to know who is present in the activespace. Informer service sends messages when persons enter or exit an activespace. These messages can be used by other services that set the user preferences or space sharing attributes.

All these messages are sent as ‘events’ using the OMG Event Service. Different event channels are used for sending different kinds of events.

3. Architecture

The Discovery service consists of several components and also depends on some external components. We now describe these components.

3.1 Entity

Every entity should have the capability to send heartbeats at regular intervals. Code was added to Entity_i for this purpose. Every entity inherits Entity_i, and so, the functionality to send heartbeats is also inherited. Each heartbeat contains information identifying the entity. This includes the name, type of entity, the UCR (Unified Component Reference) and the duration for which this heartbeat is valid. These heartbeats are sent in the form of events on the PRESENCE channel. There are separate SERVICE PRESENCE and DEVICE PRESENCE channels for services and devices respectively. Each heartbeat contains following information:

• string name	: The name of the entity
• string type	: The type, service or device
• string UCR	: The Unified Component Reference
• long validDuration	: Duration of validity of this heartbeat

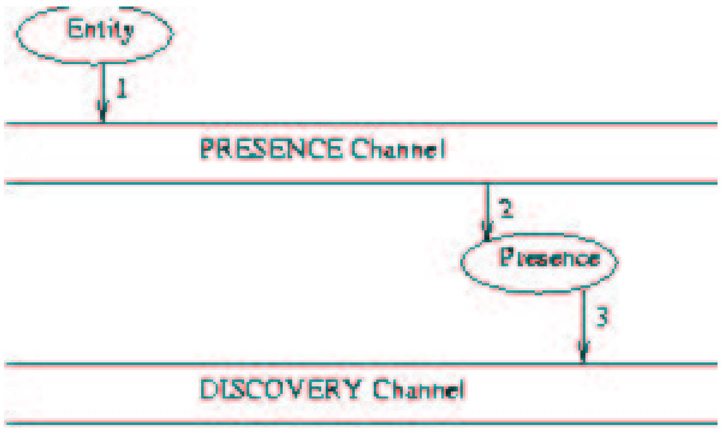
3.2 Leasing Mechanism

Presence service keeps track of entities by taking leases on behalf of an entity whenever a heartbeat is received. Informer service also takes similar leases on behalf of persons. Thus we have a separate leasing mechanism that handles the leasing functionality. The same leasing mechanism is used by both the services.

To use the leasing mechanism, an object should create an instance of the leaseframework, and register with it. Now, the object can take leases on behalf of client objects, specifying a duration for which the lease should be valid. When a new lease is taken, the object gets a notification that a new lease has been taken. This is done by a function call on the object. Similarly, the object also receives a notification when an existing lease expires. There is no notification when existing leases are renewed.

3.3 Presence

Presence service listens to the PRESENCE channel of services and devices, for heartbeats of entities. When it receives a heartbeat, it takes a lease for that entity. The duration of validity of this heartbeat is taken from the heartbeat. When presence gets a notification that a new lease has been granted, it sends an event on the DISCOVERY channel corresponding to that entity type (service or device), announcing the starting of this entity. If an entity stops sending heartbeats, its lease will expire, and so presence will get a notification saying that a lease has expired. Presence interprets this to mean that the entity has stopped, and thereby sends an event on the DISCOVERY channel saying that the entity has stopped. Thus, if an entity crashes, or is stopped because of some other reason, Presence service detects that and sends an event. This can be used to make the system more stable by restarting crashed components.



The Figure shows the steps involved:

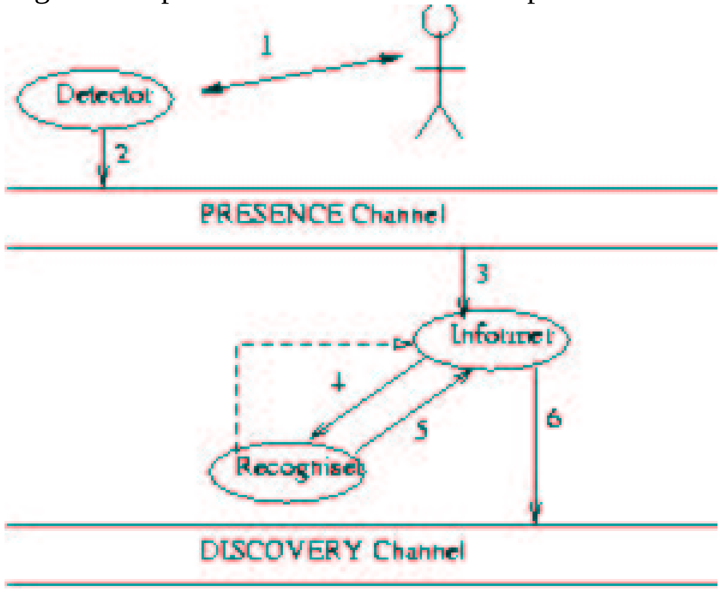
1. Entity sends heartbeat on presence channel.
2. Presence service gets the heartbeat event.
3. After processing the heartbeat, Presence service sends an event on discovery channel if an entity starts or stops.

3.4 Informer

The Informer service is similar to the Presence service. While presence service keeps track of entities in an activespace, informer service keeps track of persons present in an activespace. Entities are components that run in a computer, and so, can send CORBA events declaring their presence. This cannot be extended to persons. We need some detection devices to detect the presence of persons in an activespace. These detection devices can detect persons present in the activespace at any time. Thus, such a detection device can periodically report its observations.

The observations of detecting devices are sent as events on the PERSON PRESENCE channel for persons. Informer listens to these events to find out which persons are present in the activespace. There can be many different kinds of detecting devices. With time, new detecting devices can also be added to activespaces. The format of observations may be different for different devices. Informer cannot be expected to understand all such observation events. Thus, we have a Recogniser for every kind of detecting device. The recogniser for a particular detecting device should be able to interpret the observation event sent by the device, and convert it to a list of persons detected. Thus, on getting an observation event, Informer calls the corresponding recogniser to find out which persons have been detected. Informer then takes leases for each person. When informer gets a notification that a new lease has been granted, it sends an event on the PERSON

DISCOVERY channel for persons announcing that the person has entered the activespace. Similarly, when informer gets a notification that a lease has expired, it sends an event announcing that the person has exited the activespace.



The Figure shows the steps involved:

1. Detector detects person.
2. Detector sends observation event on presence channel.
3. Informer service gets the observation event.
4. Informer calls the Recogniser for this kind of observation event.
5. Recogniser returns information about persons detected.
6. Informer processes this information and sends an event on the discovery channel if a person enters or exits the activespace.

3.5 Recognisers

As mentioned earlier, there is a separate recogniser for interpreting the observations of every kind of detector. We need recognisers since the detectors themselves cannot be expected to interpret their observations. The interpretation might include things like authentication, depending on the settings of the activespace. The detecting device does not know about such policies, and so, reports its own observations only; it does not try to interpret them.

Every recogniser hooks itself to the Informer service when it is started. Whenever Informer gets an observation event, it calls the appropriate recogniser. Each observation event contains its type, and so we can find out which recogniser is to be called. This method provides a good decoupling of interpreting observations and the actual tracking of persons. The Informer does not need to know about the different detectors. Thus, new detectors can be easily added to the system without having to change the informer. At present we are using AirID Lt badge detectors, and so there is a recogniser for that.

4. Programming Entities

All service and device components should be entities. Things to be done to achieve this are:

- The component should inherit from `Entity_i` (instead of `BaseComponent` or `CORBAComponent_i`).
- The `init()` method should initialize the entity by calling `entityInitialze()`. This function returns 1 on success and –1 on failure, and takes 4 arguments, the last one being optional. The arguments are:
  - `char * name`: The name of this component
  - `char * type`: The type of this component, which should be either `"SERVICE"` or `"DEVICE"`. This decides which PRESENCE channel to send the heartbeats on.
  - `char * activeSpaceName`: The name of the activespace this component is present in.
  - `long heartbeatPeriod`: The period (in seconds) for sending heartbeats. This is an optional argument, and can be left out so that the default value is used.

  This should typically be called at the beginning of the `init()` function.
- The `main()` method should start a thread that keeps sending heartbeats periodically. This is done by calling the function `startHeartbeats()`. This function takes no arguments, and returns an `int`. The return value is 1 on success, and –1 on failure. This should typically be called at the beginning of the main`()` function.
- The `finish()` method should end the entity by calling `entityFinish()`. This function stops the thread sending heartbeats, and returns 1 on success and –1 on failure.

The 4 arguments given in `entityInitialize()`, namely `name, type, activeSpaceName` and `heartbeatPeriod` are stored as protected instance variables with the same names. So, any part of the component needing to use this information should do so using these variables.

In addition, there is a convenience function in Entity_i, which can be used by the inheriting component. It is:

- `CosNaming::NamingContext_var getNamingContext(void)` : This function returns the naming context of the activespace to which this entity belongs.

The naming context is also available in the instance variable n`amingContext`, of `Entity_i.`

A sample entity shows how this can be done:

```
-----------------------SampleEntity.h-----------------------------

#include "Entities/Entity/Entity_i.h"

class SampleEntity : public virtual Entity_i
{
public:
      int init(int argc, char ** argv);
      void main( void );
      void finish( void );
};

--------------------------------------------------------------------

-----------------------SampleEntity.cpp-----------------------------

#include "SampleEntity.h"
#include "OSDependent/OS.h"

int SampleEntity::init (int argc, char ** argv)
{
      int success;

      success = entityInitialize( "SampleEntity", "SERVICE",
                                    argv[0], 10 );
      if (success == -1) return –1;

      // all code specific to the entity goes here

      return;
}

void SampleEntity::main( void )
{
      startHeartbeats();

      // all code specific to the entity goes here

      return;
}

void SampleEntity::finish ( void )
{
      entityFinish();

      // all code specific to the entity goes here

      return;
}

COMPONENT_FACTORY_DECLARE(SampleEntity)

--------------------------------------------------------------------
```

5. Programming Recognisers

A recogniser is needed to interpret the observations of each detecting device. Every time a new detector is added, the corresponding recogniser has to be started. Informer can then contact this recogniser to interpret observations and know which persons have been detected by the detector. The following requirements have to be followed for writing recognisers:

- ➢ The component should inherit from `Recogniser`, which in turn inherits from `BaseComponent`.
- ➢ It should implement the following function:

  ```
       Gaia::Discovery::Recognised recognise( const
            Gaia::Events::EventInfo & evinfo );
  ```

  The return value is a structure containing

  - ▪ long validDuration : Duration of validity of this observation.
  - ▪ PersonList list : List of persons detected.

- ➢ It should hook itself to the Informer, the hook name being the same as the `EventId` of the observation event sent by the corresponding detector. In order to be able to hook itself to the Informer, the recogniser should be started in the same `ComponentContainer`.

The recogniser may need to contact other services such as authentication service, in order to do its duty.