Voyager™ Sample Design

Version 1.0 for Voyager 8.0

Table of Contents

Table of Figures

Introduction

Audience

This document addresses distributed system architects, designers, and implementers. The early sections describe an approach to the problem solution, while the later sections describe how some of Voyager*'s* features support or implement elements of the design. The design sections contain brief examples of how to use applicable Voyager features in Java. The .NET code to utilize those Voyager features is virtually identical to the Java code.

Prerequisites

Voyager Design Guide assumes a basic working knowledge of Java and Voyager, as well as experience developing network Java applications. Elements of the architecture and design assume Voyager 8 or later. Familiarity with the Unified Modeling Language (UML), applied to distributed system architecture and design, is also beneficial.

Overview

This document describes a problem, an architecture that describes a family of agent solutions, and a high-level agent-based design realizing the architecture. The design emphasizes use of Voyager features, explaining in each case how the Voyager feature solves a design requirement. This document contains neither a comprehensive description of the problem, a full architecture addressing the problem, nor a design expanding every element of the architecture. The architecture addresses interesting elements of the problem suitable for solving using a solution incorporating autonomous agents, a design that expands on the agents' responsibilities, and commentary linking features of Voyager that support implementation of the design.

Contacting Technical Support

Recursion Software welcomes your problem reports, and appreciates all comments and suggestions for improving Voyager. Please send all feedback to the Recursion Software Technical Support department via the email or phone at psupport@recursionsw.com or by calling (972) 731-8800.

The Problem

We want to automate a building to minimize power consumption without decreasing the occupants' comfort. In scope is control of the air conditioning and heating; lighting; and major appliances such as the water heater and dishwasher. Other facets of building management, such as intrusion detection, fire detection, flood detection, control of window shades, etc., are out of scope to simplify the example problem.

Vocabulary

The term “device” refers to something that communicates with the system and whose state is not normally changed by a person. A device, for example, would be a temperature sensor, a hot water heater, a fan motor, or an air vent damper.

The term “appliance” refers to something that communicates with the system and whose state is normally changed by a person. The list of appliances includes dishwashers, washing machines, clothes dryers, electric lights, microwave ovens, and thermostats. An appliance is considered a device with expanded capabilities.

The term “building” refers to an enclosed freestanding structure. To be managed the building must contain at least heating, air-conditioning system, and thermostat appliances.

The term “room” refers to a distinct subdivision of a building, including foyers, hallways, utility closets, or any other subdivision of the building whose environment is instrumented and controlled.

The Architecture

Use Cases

The use cases document, for each actor involved, the actions the actor initiates or receives output from the system. The four actors are described below.

- System Manager – The System Manager is responsible for configuring and monitoring the system.

- Occupant – The Occupant operates appliances, e.g., changing an appliance's state; causes changes in the environment by emitting heat; and is the beneficiary of the heating and cooling systems.

- Clock – The Clock initiates periodic events, such as the end of the day, sunrise, sunset, etc. The Clock also maintains a collection of pending device commands, ordered by the time at which the command should be issued.

- Switch – The Switch is a device that signals a change of state, e.g., when an exterior door is opened, an air conditioning fan motor turns on, or a thermostat crosses a temperature threshold setting, the Switch sends a message announcing the state change event. The event message contains the Switch's identity, type, and the nature of the state change.

System Manager Use Cases

In the following use cases the initiator is the System Manager.

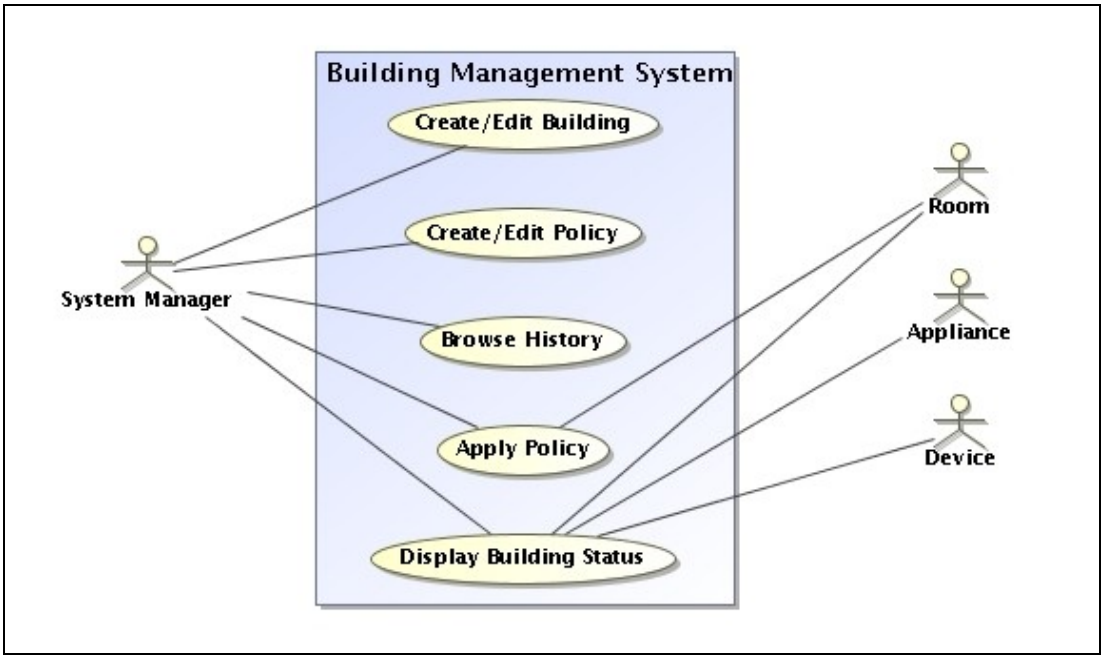

Figure 1*: System Manager Use Case*

Create/Edit Building

Summary: Create or edit the description of the building, including defining the building's rooms and the fixed devices and appliances associated with each room.

Actor: System Manager

Precondition: The system is installed and operational.

Description:

1. The System Manager logs onto the system.
2. The System Manager selects the create/edit building option.
3. The System displays the current building configuration.
4. The System Manager can elect to create or edit a room.
5. The System Manager can elect to create or edit a device.
6. The System Manager can elect to create or edit an appliance.
7. The System Manager can elect to associate a device with a room.

Postcondition: The building configuration is added to or edited.

This use case describes a classic create, read, update, and delete application. The solution could be implemented using a command language on a textual device, or as a graphical tool, the latter being the preferred approach. The solution requires access to a stored configuration, and the ability to discover and query devices present in the building.

Create/Edit Policy

Summary: Create or edit the policies describing operation of the building.

Actor: System Manager

Precondition: The system is installed, operational, and at least one room is defined.

Description:

1. The System Manager logs onto the system.
2. The System Manager selects the create/edit policy option.
3. The system displays existing policies, if any.
4. The System Manager selects an existing policy as the starting point for a new one, or elects to start with a default policy.
5. The System Manager defines the days and time of day to which the policy element applies, then defines the value or function associated with the time period.
6. A policy can also be composed of other policies. If the System Manager composes the new policy, the System Manager will be asked to resolve any conflicts that arise, such as two of the included policies applicable time periods apply simultaneously.
7. The System Manager repeats the definition action for each time period.

Postcondition: The System contains one or more valid policies.

This use case describes another classic create, read, update, and delete application.

Apply Policy

Summary: Associate one or more policies with each room.

Actor: System Manager

Precondition: At least one room and one policy are defined.

Description:

1. The building manager logs onto the system.
2. The building manager selects the Apply Policy option.
3. The system displays lists of available Rooms and available Policies.
4. Selecting a Room displays the current policies applied to selected Room.
5. Selecting a Policy displays the rooms to which the policy currently applies.
6. The manager can select a Policy currently applied to a Room and “unapply” it.
7. The manager can select a Policy and add it to the collection of Policies applied to the Room.

Postcondition: At least one Policy has been applied to each Room.

This use case describes another classic create, read, update, and delete application, this time focusing on creation and maintenance of relationships among other configuration elements.

Display Building Status

Summary: Display the status of the building.

Actor: System Manager

Precondition: The building configuration has been defined.

Description:

1. The System Manager logs onto the system.
2. The System Manager selects the Show Status option.
3. The system displays the summary building status and a list of defined Rooms.
4. Selecting a Room displays the current status of the selected Room.
5. The system updates the status display to keep it consistent with the state of the displayed devices.
6. The System Manager can elect to jump directly from a Room status display to the Apply Policy function.

Postcondition: The status of the building or room is displayed and automatically refreshed as needed.

This use case describes a query application with a requirement to continuously update the display.

Browse History

Summary: Display the message traffic history.

Actor: System Manager

Precondition: The System has been started at least once.

Description:

1. The System Manager logs onto the system.
2. The System Manager selects the Browse History option.
3. The System Manager forms a query describing the date range of interest, the type of messages to include, etc., and sends the query.
4. The system executes the query to retrieve the messages and displays the result.

Postcondition: The status of the building or room is displayed.

This use case describes a query application with a requirement browse what could be a very large collection of records. This use case also implies a requirement to capture and retain an event history.

Occupant Use Cases

In each of the following use cases the Occupant is the initiator.

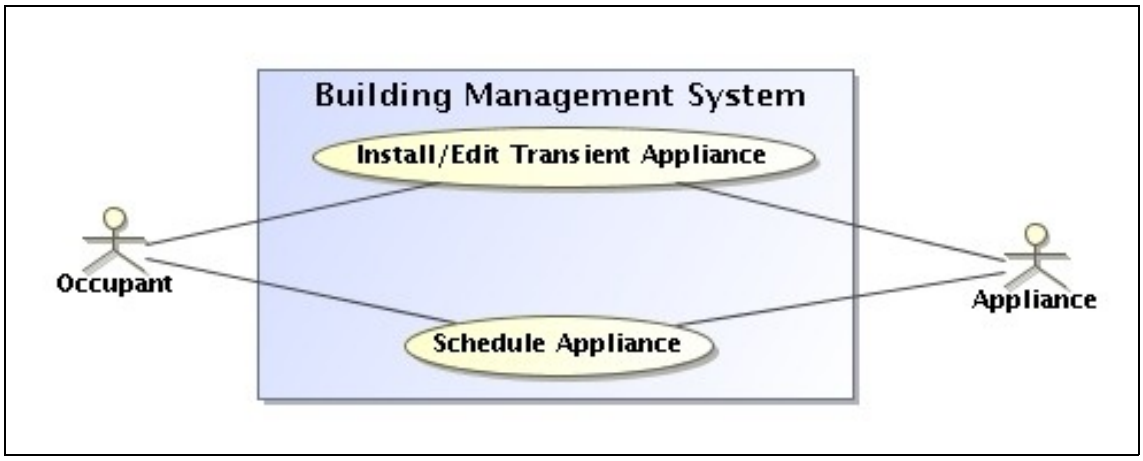

Figure 2: Occupant Use Cases

Install/Edit Transient Appliance

Summary: Install a transient appliance, i.e., an appliance that can be moved from room to room.

Actor: Occupant

Precondition: The appliance is not installed in the building and not communicating with the system.

Description:

1. The occupant connects the appliance to the building.
2. The appliance broadcasts a query asking to connect to the system.
3. The Building Manager Agent receives the query.
4. If the appliance successfully connected to the system at least once, the Building Manager Agent updates the building configuration and sends the appliance a welcome message whose content enables the appliance to complete the connection process.
5. If the appliance has never connected to the building, the Building Manager Agent negotiates with the appliance to determine the appliance attributes, updates the building configuration with the appliance's description and configuration, and completes the configuration conversation with a welcome message whose content enables the appliance to complete the connection process.

6. The appliance receives the Building Manager Agent's welcome message, updates the appliance configuration, and enters normal operation state.

Postcondition: The appliance is communicating with the system and the appliance's description is entered into the building's configuration.

Schedule Appliance

Summary: Schedule operation of an appliance.

Actor: Occupant

Precondition: The appliance is operational.

Description:

1. The Occupant selects options on the appliance front panel, including immediate or deferred start.
2. The appliance posts a message asking that the appliance's operation be scheduled. The message includes the requested appliance configuration, how long the operation will take, and how much of what resources the appliance will require.
3. The Room Manager Agent responsible for the room in which the appliance is installed reads the posted message, mediates among the interested Agents to arrive at an acceptable schedule, then posts a resource schedule request message describing the schedule event.
4. The Power Management Agent receives the Room Manager Agent's resource schedule request, estimates the request's resource consumption and replies with a recommended schedule.
5. The Temperature Management Agent receives the Room Manager Agent's resource schedule request, estimates the change in the room's environment, estimates the resources needed to maintain the room within environmental limits, and replies with a message containing a recommended schedule.
6. Based on the responses from the Power Management Agent and Temperature Management Agent, the Room Manager Agent creates a start message, wraps the start message in a schedule request and sends the schedule request to the Clock and the appliance.
7. The Clock agent reads the Room Manager's schedule request message describing the event and adds the request to the Clock's schedule.
8. The appliance reads the Room Manager's message and displays on its panel an indication that the operation has been scheduled.
9. If the Room Manager cannot arrive at an acceptable schedule, it posts a message denying the scheduling request. If this happens, the appliance responds to the denial by either changing a parameter to reduce resource consumption or changing the time window of the request and resubmit the

request. A final fallback, which is possible only for appliances with an appropriate interface, is to display an error message informing the user that the start request, as currently configured, has been denied.

Postcondition: Operation of the appliance has been scheduled.

Clock Use Cases

In each of the following use cases the Occupant is the initiator. Figure 3 shows a graphical representation of the two use cases.

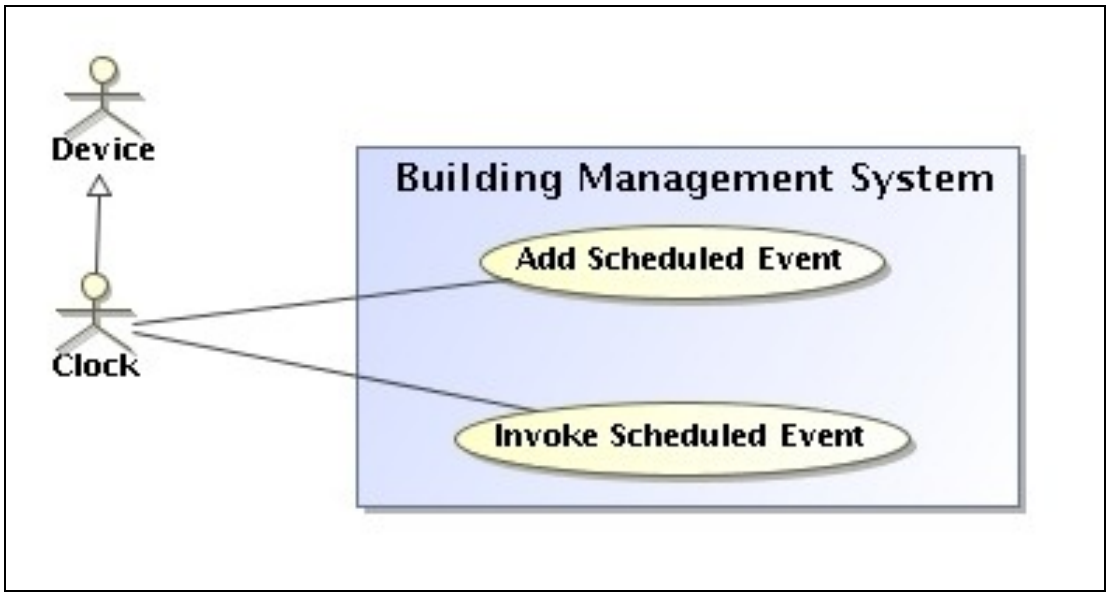

Figure 3: Clock Use Cases

Add Scheduled Event

Summary: Transmit Schedule Event

Actor: Clock

Precondition: The event to be scheduled has been defined.

Description:

1. The Clock Agent receives a request to schedule an event.
2. The Clock Agent validates the event.
3. If the validation fails, the Clock Agent sends a negative acknowledgment accepting the event.
4. If validation succeeds the Clock Agent sends the scheduled event to reliable storage.
5. The Clock Agent adds the event to the time ordered collection of pending events.

6. The Clock Agent sends an acknowledgment accepting the event.

Postcondition: The valid event message was stored and inserted into the schedule.

Invoke Scheduled Event

Summary: Post a previously scheduled event message.

Actor: Clock Agent

Precondition: The current date and time of day match the next scheduled event.

Description:

1. The Clock Agent retrieves the scheduled event from reliable storage.
2. The Clock Agent posts the event.
3. The Clock Agent removes the scheduled from the current schedule.
4. The device or appliance that is the subject of the event reads the message and takes the action described by the event message.

Postcondition: The event message was posted and the pending event database updated.

Thermostat Use Cases

In each of the following use cases the Occupant is the initiator.

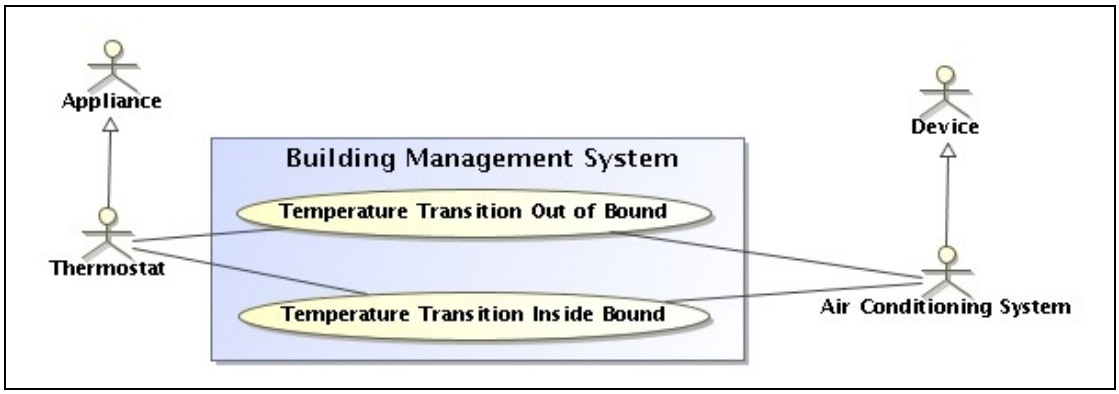

Figure 4: Thermostat Use Cases

Temperature Transition Out of Bound

Summary: A thermostat exception event is signaled when a room's temperature moves from an acceptable to unacceptable value, e.g., a room's temperature rises above the thermostat's high temperature limit setting.

Actor: Thermostat

Precondition: The Room is configured with a thermostat and the thermostat is configured with temperature limits and mode (heating or cooling).

Description:

1. The thermostat posts a message describing the event, including the current and limit temperatures.
2. The Temperature Management Agent reads the event.
3. The Temperature Management Agent treats the event as a request for resources and processes the event in the context of the current state of adjacent rooms, the outside temperature, the current state of the air conditioning system, and the current temperature policy.
4. The Temperature Management Agent creates and posts a message requesting scheduling of the power needed to implement the temperature change. The total amount of power required is the sum of the power required by the air conditioning compressor, the air handler fan, and the vent dampers.
5. The Power Management Agent receives the Room Manager Agent's resource schedule request, estimates the request's resource consumption and replies with a recommended schedule. When determining the recommended schedule the Power Management Agent considers the size of the requested allocation, the power policy in effect at the start of consumption, and the expected total power consumption at the start of consumption.
6. If the policies are satisfied and the Power Management Agent provides an acceptable schedule, the Temperature Management Agent posts a message requesting scheduling of the air conditioning, fan, and damper device state changes.
7. The Clock agent reads the Room Manager's schedule request message describing the event and adds the request to the Clock's schedule.

Postcondition: The air conditioning appliance state change is scheduled.

Temperature Transition Inside Bound

Summary: A thermostat switch is tripped when a room's temperature crosses a predefined threshold, e.g., a room's temperature drops below the thermostat's high temperature setting.

Actor: Thermostat

Precondition: The Room is configured with a thermostat and the thermostat is configured with temperature limits and mode (heating or cooling).

Description:

1. The thermostat posts a message describing the event, including the current and limit temperatures.
2. The Temperature Management Agent reads the event.
3. The Temperature Management Agent treats the event as a request for resources and processes the event in the context of the current state of

adjacent rooms, the outside temperature, the current state of the air conditioning system, and the current temperature policy.

4. The Temperature Management Agent creates and posts a message requesting scheduling of the power needed to implement the temperature change.
5. The Power Management Agent receives the Room Manager Agent's resource schedule request, estimates the request's resource consumption and replies with a recommended schedule. When determining the recommended schedule the Power Management Agent considers the size of the requested allocation, the power policy in effect at the start of consumption, and the expected total power consumption at the start of consumption.
6. If the policies are satisfied and the Power Management Agent provides an acceptable schedule, the Temperature Management Agent posts a message requesting scheduling of the air conditioning appliance state change.
7. The Clock agent reads the Room Manager's schedule request message describing the event and adds the request to the Clock's schedule.

Postcondition: The air conditioning appliance state change is scheduled.

Static Model

The static model shows, at a high level, the objects mentioned in the use cases and their relationships. The static model here is expressed as a collection of interfaces, both to emphasize the high level nature of the model and because Voyager works best in terms of interfaces.

The static model has been divided into two parts. The first part, Figure 5, is the static model of the passive objects, i.e., the classes without their own process thread. The second part, Figure 6, is the static model of the active objects, i.e., the agents, each of which executes on its own thread.

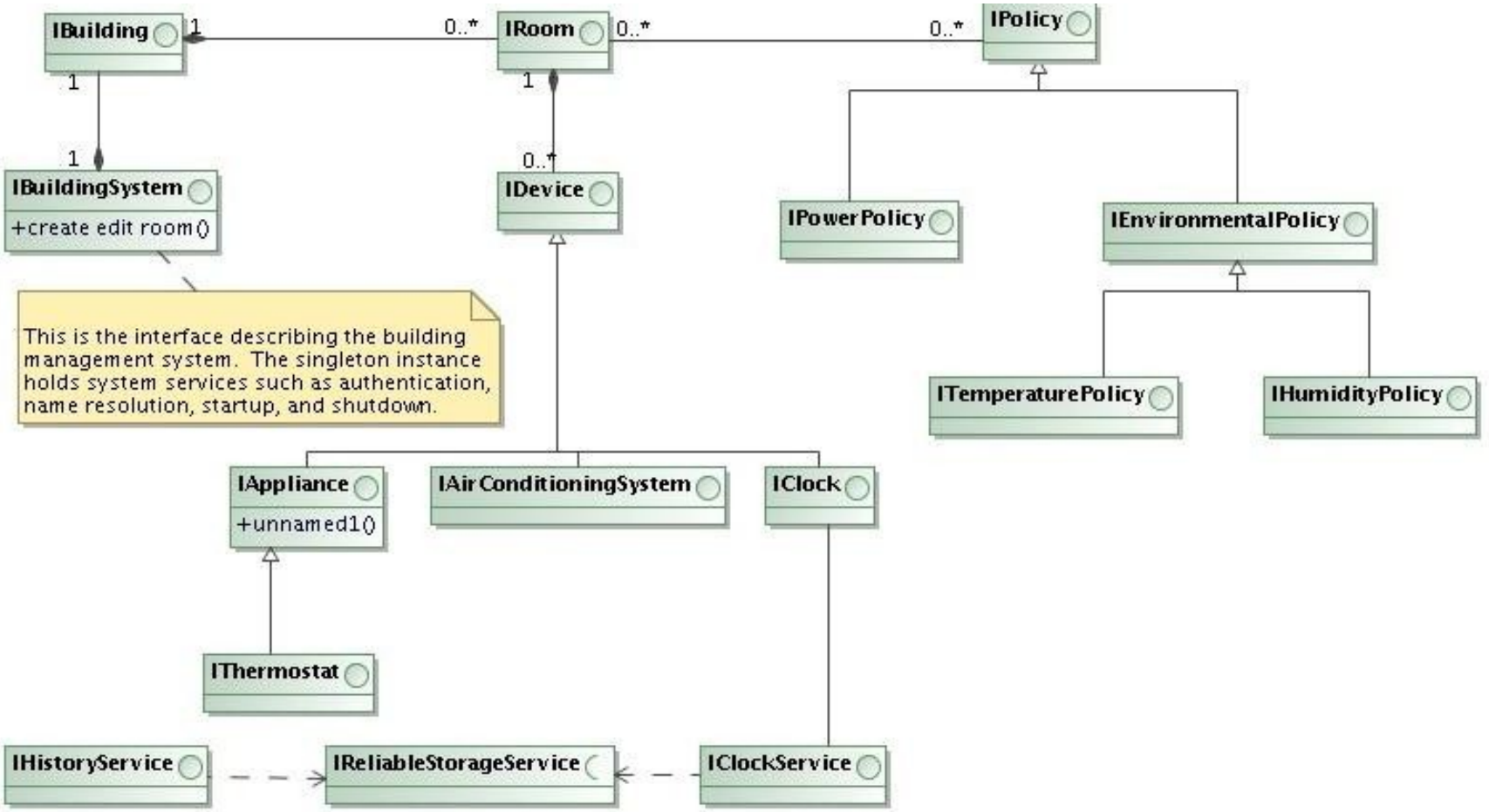

Figure 5: The Static Model for Passive Objects

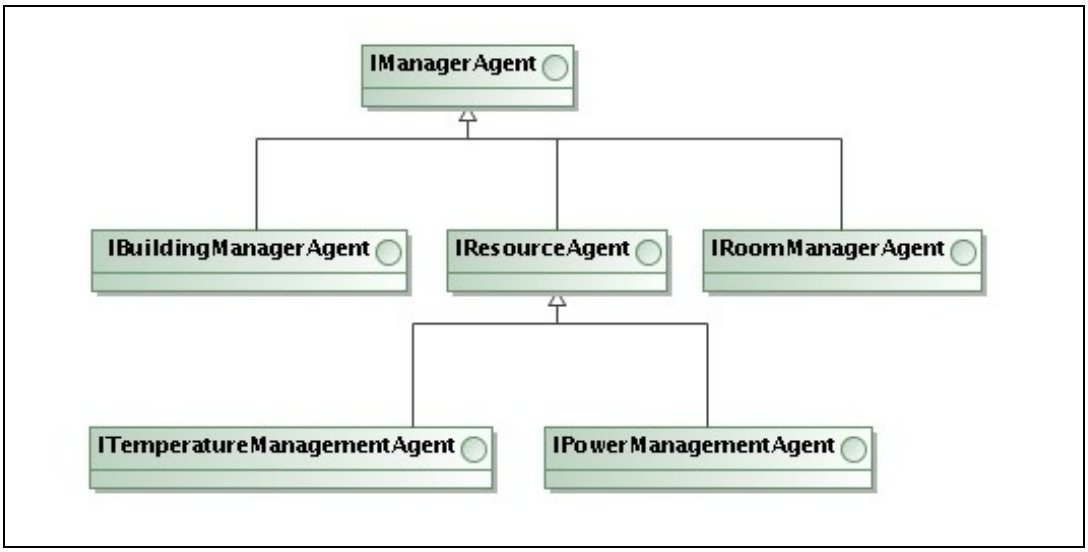

Figure 6: The Static Model for Active Objects (Agents)

Object Interactions

The following object interaction diagrams show, as sequence diagrams, the messages mentioned in selected use cases. In particular, in more than one case several objects need the same message. The diagrams point out which messages need to be read by more than one receiver.

Create/Edit Building Interaction Diagram

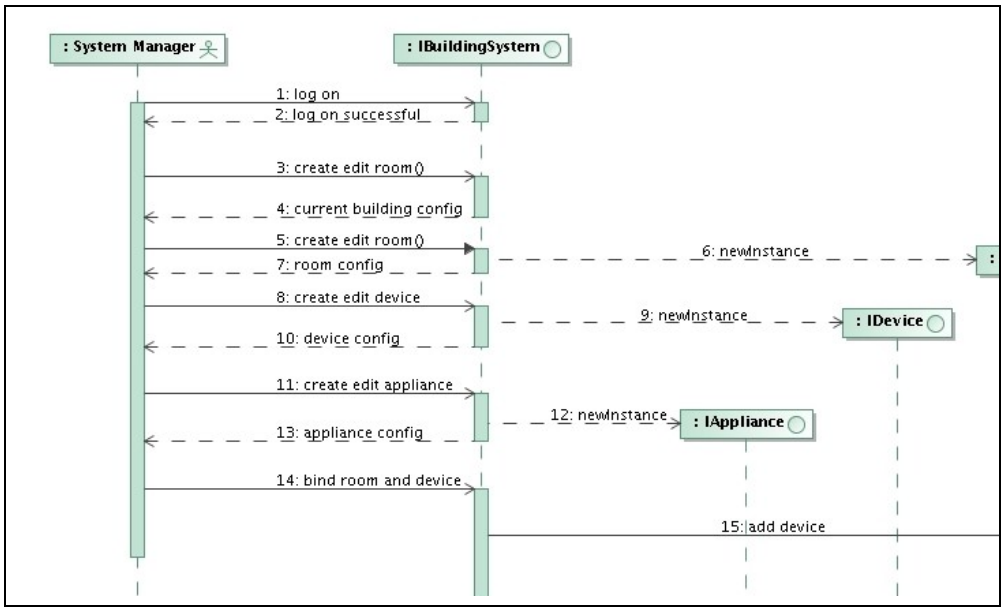

Figure 7: Create/Edit Building Interaction Diagram

Schedule Appliance Interaction Diagram

Note in this diagram that in two cases the identical message is sent to multiple recipients. This suggests some sort of publish/subscribe or multicast solution.

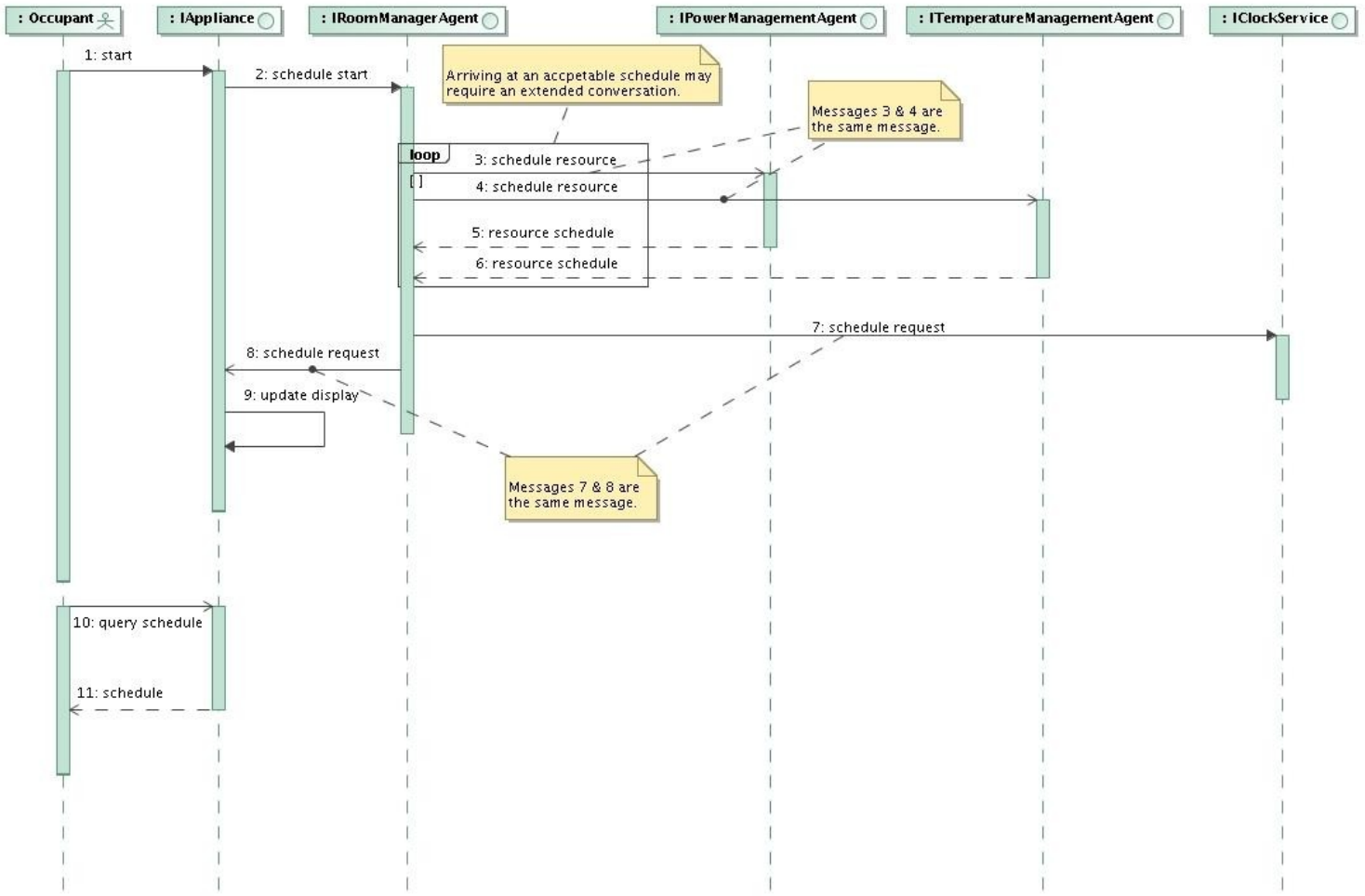

Figure 8: Schedule Appliance Interaction Diagram

The High Level Design

The following sections describe the high level design.

Assumptions

1. All devices and appliances of interest reliably communicate with the System.

2. Every room contains at least a thermometer and a heating and air conditioning vent.

3. All appliances offer a software interface supporting query of the equipment's state and at least the same controls the equipment presents to the human user. In other words, a dishwasher presents to the application an interface duplicating the status and controls found on the dishwasher's front panel, plus additional information such as the amount of water, power, and time required for operation using the current configuration.

4. All devices offer a software interface supporting query of the equipment's state.

5. The building is managed by a single entity, eliminating any requirement to support more than one point of administrative control.

Devices

A Device represents something of interest in a room that communicates. A Device might be a microswitch, thermometer, humidistat, or light sensor, i.e., a device that provides status. A Device might also be a light switch, an air conditioning vent damper, a door lock, or a water valve, i.e., something that not only provides status but that can also accept commands to change state.

Services

The design incorporates a number of services, where a “service” accepts and acts on requests, but unlike an agent, cannot initiate activity because it does not contain its own thread. A service normally exports an interface, and frequently registers itself in a directory.

Directory Services

The solution requires two different directory services, both of which Voyager provides.

The White Pages Service maps a name to an object. A white pages lookup is usually for an exact match. The Voyager white pages, accessed using the methods found in

com.recursionsw.ve.Namespace, provides directory entry life cycle as well as lookup services. A white pages lookup query always returns either exactly one match or a **com.recursionsw.ve.NamespaceException** exception.

If the object implementing the **IBuilding** interface registers itself with the White Pages Service under the name "TheBuilding", then a client application, such as the one used in the by the System Administrator in the Create/Edit Building use case, finds the building object using code that looks like the following lines, assuming the Voyager directory service is referenced by the **ClientContext** named **directoryServer**.

```
Namespace whitepages = directoryServer.getNamespace();

IBuilding market = (IBuilding)whitepages.lookup("/TheBuilding" );
```

The Voyager Yellow Pages Directory provides a mapping between a *service description*, consisting of one or more name-value *service attribute*s, and a service. A Yellow Pages lookup is performed using a *discovery request* containing an expression to match against service descriptions. The Yellow Pages Directory returns all service descriptions that match the expression in the discovery request. A client performing a Yellow Pages lookup is asking for all the services that match a filter: the discovery request expression.

Some buildings contain more than one air conditioning system. If each air conditioning object, all of which must implement the **IAirConditioningSystem** interface, register themselves with the Yellow Pages Service with a service description containing the attribute name "HVAC" then a class implementing the **ITemperatureManagementAgent** interface can find all the air conditioning systems in the building using the following code.

```
// assume "directoryServer" is the ClientContext referencing

// the Voyager hosting the Yellow Pages directory service.

DiscoveryRequest request = new DiscoveryRequest();
DiscoveryRequestExpression expr = new
DiscoveryRequestExpression();

expr.add(ExpressionFactory.exists("HVAC"));

request.setRequestExpression(expr);

IYellowPages yp = YellowPages.getInstance(directoryServer);

ServiceDescription[] serviceDesc = yp.lookup(discoveryRequest);

IAirConditioningSystem agent0 = (IAirConditioningSystem)
serviceDesc[0].resolveService();
```

The **serviceDesc** array contains the query result, and assigning a value to the **agent0** variable illustrates retrieving a reference to the first **IAirConditioningSystem** instance the query found.

Reliable Store Service

The Reliable Store Service provides stable storage for long-lived information about the system, such as the building configuration, the message history documenting the changes in the building's state, and the Clock Service's collection of scheduled events.

Voyager provides a convenient API for accessing databases directly through JDBC. The **Voyager Database Developer's Guide** documents the interface. A Reliable Store Service is readily built on this API.

Policies

Policies represent the preferred state of a resource over time. An electrical power policy might prefer minimum power consumption on weekends, holidays, and on workdays between the hours of 7 p.m. and 7 a.m. A temperature policy might prefer a warmer temperature during non-work hours in summer and a cooler temperature during non-work hours in winter. When the cost of a resource varies by time of day or season, a policy considers the varying cost when choosing a preference.

As the examples illustrate, policies express a preference based on time of day, day of the week, or any other property associated with the calendar.

A Voyager agent wrapping a rule engine is a reasonable approach to implementing a flexible policy mechanism, and more flexible than procedural code.

Resources

A Resource is something measurable that has an associated cost. Some resource consumption can be scheduled, such as when to turn on an air conditioning compressor or water heater. Other resource consumption is unconditional, such as when an occupant turns on a television.

Resources of interest, i.e., that can be measured and managed in this System, include electricity and room temperature. Consuming some resources requires consumption of others, e.g., turning on an air conditioning compressor and fan requires consuming some amount of the electricity resource for a span of time.

Resources know the cost of consumption a unit of the resource, e.g., the cost of a kilowatt-hour at a time of day, the consumption in kilowatts of running an air conditioning compressor, the cost of changing the temperature of a space by one degree (which is actually the sum the resource consumption of several devices).

A Policy can describe a preference for when a resource should be consumed, if the consumption can be scheduled; a limit on resource consumption; or a trade off applied when a resource's consumption must be reduced.

Agent Types

Building Manager Agent

The Building Manager knows about the all the Rooms that together define the building.

The Building Manager's responsibilities include the following.

- Providing a one-to-many communication mechanism for use by agents operating on behalf of the Building Manager, and by the Room Managers for which the Building Manager is responsible.
- Providing well-known directory services. The provided directory services include lookups by identifier, usually referred to as white pages, as well as lookups by property, usually referred to as yellow pages.
- Providing notifications to all Rooms, such as in an emergency. Conversely, the Building Manager listens for and responds to exceptional conditions a Room Manager might detect, such as excessive heat, unexpected water, or an unscheduled and unauthorized access to a restricted area of the building.
- Mediating conflicts among Room Managers, e.g., when the total of the Room Managers' requests for air conditioning exceed the capacity of the building's air conditioning equipment the Building Manager allocates the air conditioning resource equitably among the Room Managers.
- Providing interfaces to the outside world, such as outdoor air temperature, precipitation, emergency notifications, etc. The Building Manager periodically publishes the current values of real world sensors, such as the current outside temperature.

Room Manager Agent

The Room Manager represents some sort of manageable space, such as a room, hallway, outdoor patio, etc. The Room Manager adds itself to the white pages directory. The Room Manager knows what Devices populate a Room, and the Policies that apply to the space. The Room Manager is responsible for providing a one-to-many communication mechanism for use by agents operating on behalf of the Room Manager.

Device Agent

A Device Agent provides a uniform interface for one Device or Appliance and implements either **`IDevice`** or **`IAppliance.`** A Device Agent is responsible for translating between the ontology (vocabulary) used by the agents and the device's native language. If the agents ask for a change to “on” state by saying “action enable” and the device accepts the command to turn on by accepting the command “set power 1”, the Device Agent is responsible for translating the command. Similarly, if a device expresses a temperature in degrees Fahrenheit and the system operates in Celsius, the Device Agent is responsible for translating what the device says into the unit of measure the system expects.

If the Device publishes asynchronous events, such as an alarm or a periodic status update, the Device Agent is responsible for forming and posting an appropriate message.

A Device Agent may also operate at a higher level as a gateway between Devices.

Clock Agent

The Clock Agent accepts requests to schedule message publication, saves future publication events using the Reliable Store Service, and waits for the current time to match the time of the next publication event. When the current time matches a publication event's time, the Clock Agent publishes the message associated with the publication event.

Historian

The Historian creates a permanent record of the message traffic in the system, suitable for reporting resource consumption over a particular time window; maximum, mean, and minimum values for a particular sensor's value; the number of times an event occurred, such as the number of times a door opened and closed over a weekend; etc.

System Manager

The System Manager agent starts and stops the system; handles the arrival of new Devices and the departure of existing Devices; and responds to changes in the system's environment such as the change in the cost of a consuming a resource.

Message Types

Messages among the agents will implement the semantics of the FIPA Communicative Act Library Specification[1]. However, the messages will be formatted as Java class instances, not in the text format found in the FIPA specification.

Adopting the FIPA Communicative Act model means, for example, that sending to anyone interested the current state of a thermometer is an "inform" message, and changing the state of a Device is a "propose" message from the requester and an "accept" or "refuse" response from the Device.

The system requires the following message types.

- Accept Proposal – Accept a previous proposal to take some action.
- Agree – Agree to perform some action, possibly at a future time
- Cancel – A request from one agent to another asking the other agent to not carry out a previously agreed action.

1 Published by the Foundation for Intelligent Physical Agents at http://www.fipa.org/specs/fipa00037/SC00037J.html as document SC00037J.

- Call for Proposal – A request for proposals to carry out an action.
- Failure – An agent failed to complete an action.
- Inform – An agent's assertion of a fact.
- Not Understood – An agent's response when a message is not understood by the receiver.
- Propose – An agent's proposal to carry out an action, given the appropriate preconditions are satisfied.
- Query If – A message from one agent asking another if an assertion is true or false.
- Refuse – An agent refusing to carry out a requested action.
- Reject Proposal – A refusal by an agent to carry out an action proposed during a negotiation.
- Request – One agent asking another to perform an action.
- Request When – One agent asking another to perform an action as soon as a precondition becomes true.

Communications

The use cases contain both one-to-one and one-to-many communication. The one-to-many communication is sometimes called publish/subscribe.

Voyager provides transparent one-to-one communication using a proxy for the actual object. The proxy object, which implements the interfaces the actual object implements, is responsible for translating a method call into a message, forwarding the message to the actual object, invoking the method implementation, and, optionally, returning the result to the caller. In above directory lookup examples the query usually returns a proxy to the actual object, which makes using one-to-one communication transparent to the developer.

At startup time the Building Manager Agent is responsible for creating, among other things, at least one Temperature Management Agent, which must implement the **`ITemperatureManagementAgent`** interface. The following lines of code create a Temperature Management Agent in the Voyager instance listening on port 9000 on host TemperatureAgentHost, and register it in the local White Pages Service.

```
// assume "directoryServer" is the ClientContext referencing
// the Voyager hosting the directory services
ITemperatureManagementAgent newAgent =
  (ITemperatureManagementAgent)
```

```
directoryServer.getFactory.create("TemperatureManagementAgent",
"//TemperatureAgentHost:9000" );
String agentName = newAgent.getName();
directoryServer.getNamespace().bind(agentName, newAgent );
```

In the above code the **`Factory.create()`** call returns a proxy to the instance of **`TemperatureManagementAgent`** the call builds. The **`newAgent.getName()`** call is actually a call to a remote object.

Voyager provides an easy to use and efficient one-to-many communication implementation using Spaces.

This design faces several choices for configuring the Spaces. The design could choose to use a single Space, and rely on each listener (subscriber) to filter out the uninteresting messages. This approach is attractive for the following reasons.

- The system configuration is simple. The Building Manager creates the Space and adds it to the White Pages Service under a well-known name. When other elements of the system initialize themselves they query the White Pages Service for the Space and connect to it.
- The Historian Agent is simple, since there is only a single Space to which messages are posted. A history of interesting state changes is easy to record.
- Extending the system with new kinds of agents is easy, since all agents and services interact through the Space. For example, adding a new kind of agent requires no changes to the Historian.
- Securing messages must occur at the message level, and even then security messages would be susceptible to traffic analysis.

The design could choose to segregate messages into several Spaces. A Space could be dedicated to the negotiation found in the Schedule Appliance, Temperature Transition Out Of Bound, and Temperature Transition In Bound use cases. Another Space could be dedicated to startup/shutdown, alarms, and emergency messages. Yet another Space might contain all security-related messages. Finally, one Space would be used for all other messages.

- System configuration is more complex, since responsibility for creating Spaces is not centralized.
- Isolating some message traffic, such as security messages, to a separate Space can decrease the possibility that an unauthorized agent could ease drop on message traffic.
- In a large building Spaces could be created for subdivisions of the building to reduce the amount of message traffic appearing in a single Space. For example, a Space might be created for the set of rooms sharing the same set of services, e.g., the set of rooms sharing a single air conditioning system.

If Spaces are created for subdivisions of the building, the Space architecture still allows the creation of a single Space interconnecting the subdivision spaces so that the Building Manager Agent can send a message to all subdivisions with a single method call.

Regardless of how the one-to-many Spaces are configured, using Spaces provides a simple and efficient mechanism for one-to-many communication. The following lines of code create a SubSpace and connect the new SubSpace to the Space named "BuildingGlobal".

```
// assume "directoryServer" is the ClientContext referencing
// the Voyager hosting the directory services
ISpace msgSpace = new TcpSubspace("BuildingGlobal");
ITcpSubspaceConnections neighbor = (ITcpSubspaceConnections)
  directoryServer.getNamespace.lookup("/TheBuildingSpace" );
msgSpace.connect(neighbor);
```

After the above code executes and assuming that **`msgSpace`** is in scope, the following lines will send the message built by the **`constructMsg()`** call to the topic returned by the **`getCurrentTopic()`** call.

```
Topic publishTopic = getCurrentTopic();
Publish.invoke( msgSpace, constructMsg(), publishTopic );
```

The above lines deliver the message to all listeners registered with the "BuildingGlobal" Space. The following lines register an event listener that receives the messages, assuming the listener class name is **`MsgHandler`**, which means **`MsgHandler`** implements **`PublishedEventListener`**. The **`getSubscriberTopic()`** call returns a **`Topic`** instance with an appropriate definition.

```
MsgHandler handler = new MsgHandler();
Subscriber aSub = new Subscriber( handler );
aSub.subscribe( new Topic(getSubscriberTopic()) );
msgSpace.add( aSub );
```

When a message arrives it is delivered to the listener's implementation of **`publishedEvent()`**.

The Space examples, found in the Voyager Core Developer's Guide and included in the distribution archive, provide a running example of both ways of doing publish/subscribe using Spaces.

Summary

Previous sections outlined a problem, offered an agent-based architecture meeting the problem requirements, and then sketched a high level design implementing the architecture. For some features of the design the text highlighted the Voyager features providing some or all of the implementation.

Additional Resources

Recursion Software offers additional technical guides and white papers via our website recursionsw.com.

As always, we are interested in receiving feedback about this guide or any of our products. Please contact our Engineering department to discuss any of the material presented in this document at engineering@recursionsw.com.