Middleware Lab 02 :
Ad-hoc service protocol design
and implementation

Guillaume-Jean Herbiet
<guillaume.herbiet@uni.lu>

November 7, 2008
November 21, 2008

Abstract

During the previous lab, you learned the principles of event-based discrete network simulation and became familiar with OMNeT++ and the Mobility Framework .

We ended by setting up a wireless network, using 802.11 for channel access and providing a simple broadcast application.

In this lab, you will be given the description of a more complex protocol for ad hoc services. Based on this information, you will have to design this protocol as a finite-state machine (FSM) and implement your design in OMNeT++ .

1 Instant messenging for ad hoc networks

1.1 Context

You may all be familiar with instant messenging. In the traditional (non ad hoc) approach, instant messenging clients register to a central server upon login. This server stores your current state (free, idle, busy, away, offline, etc.) and provides you with the state of the users you have placed in your contact list.

Based on this information, you are free to start chatting with some of your contacts. Improved versions of instant messenging clients also provides more advanced conversation features (smileys, videoconferencing, shared games, etc.) and allow a common conversation between more than two users.

In any case, the services provided by such an application are twofold:

- presence awareness (i.e. making your state available to others);
- conversation protocol (i.e. how to exchange information between users).

1.2 The ad hoc challenges

In an ad-hoc network, the assumption is made that no node should play a particular role (the network is said infrastructureless) and that it might be possible that the network is isolated (i.e. not connected to an external network like the Internet, for instance).

Therefore, an ad hoc service protocol for instant messaging should not rely on an external server to centralize information on users current state. This centralized scheme should also not be used to coordinate the communications between users (e.g. handle initialization and closing of a conversation between users).

For a service protocol to be compatible with ad hoc networks, all decisions should be decentralized. Each node of the network should know enough information about the network (or at least a subpart of the network) to be able to make some local decisions (e.g. consider a user as connected, start a conversation with him, etc.)

Other important points that constrain the design of ad-hoc service protocols are the fact that, as mobile wireless devices, nodes of the network are resource-limited (running on battery for instance), use a shared medium for communication (the radio medium) and are mobile (any node can break connectivity at any time due to mobility).

Finally, as a service protocol should offer some reliability to the end-user, protocol design should include robustness in its definition.

1.3 Objectives

In this lab, given the desciption of the services a protocol should offer to the end user, you will design it for usage with wireless ad hoc networks, with respect to the previously-mentioned challenges.

The description will be given as a set of requirements the protocol must respect. This approach is widely used in the industry where requirements are most of the time the expression of the customer needs.

For design, you will use the finite-state machine paradigm introduced later in this document.

Finally, you will implement your design in OMNeT++ and simulate the actual behavior of your protocol.

2 Protocol definition

As stated previously, services for instant messenging are composed of presence awareness and conversation protocols.

2.1 Presence awareness

2.1.1 Users discovery

As no centralized server can be used for presence awareness, the application protocol will rely on broadcast beacon packets for each node to announce its presence and current state:

Requirement 1

Each node shall periodically send a beacon packet, announcing a unique identifier (you can for instance use the application address provided by the Mobility Framework), its current state, and a timestamp of the packet generation date.

Requirement 2

Upon receiving of a beacon packet originating from another node, a node shall store this originator in a list of known users with its current state and an expiration time, provided this information is not outdated.

Requirement 3

To handle link breakage due to node failure and mobility, each node shall periodically remove all expired entries in its known users list.

2.1.2 State management

As in traditional instant messenging applications, this protocol will allow users to have different states. However, the states won't be decided by the user itself but will be directly managed by the protocol itself.

Requirement 4

The protocol shall start in the `INIT` state.

Requirement 5

Each time the known users list is empty (or all the entries are outdated) the host should announce it has no neighbor, stop all activity except beacon sending and processing.

Requirement 6

When the known users list is no longer empty, the host shall be considered available and resume corresponding activity.

Requirement 7

While available, the host shall initiate chat sessions with a random known neighbor, with a random bounded time between two initiation.

Requirement 8

While available, the protocol shall accept incoming chat sessions request.

Requirement 9

After being available for a given time, if no chat session request has been received nor sent, the host shall be considered idle.

Requirement 10

After being idle for a given time, if no chat session request has been received, the host shall be considered as being away.

Requirement 11

While idle, the host shall not send chat initiation requests.

Requirement 12

While idle, the host shall accept received chat sessions request.

Requirement 13

While away, upon receiving a chat session request, the host shall deny it but be considered as available again.

2.1.3 Chat sessions management

In this protocol, each user, when available, is likely to send a chat session request to a random known user. To simplify the protocol, we will assume that a given user can only have a single chat session at a time.

The chat initiation procedure is an example of *two way handshake*:

- The session initiator (caller) sends a `Call Request` message
- The callee, if it accepts the call, replies with a `Call Accept` message
- The caller, upon reception of this message then starts sending `Call Data`

For various reasons (mobility, message collision, etc.), some of those messages might get lost. Therefore it is safe to say that if the call is not established after a certain time, it should be considered as failed.

Requirement 14

Once a chat session has been initiated, the user shall no longer be considered available.

Requirement 15

Once a chat session request has been received, the user shall no longer be considered available or idle.

Requirement 16

If after a given time after sending or receiving a chat initiation request, the chat initiation procedure is not completed, the chat session shall be dropped and the user be considered available again.

Requirement 17

Upon reception of a chat acceptance message or chat data, the chat initiation timer shall be canceled and the user considered as being actively chatting.

It is also possible for a user to explicitely deny the initiation of a chat session, for instance if it is not ready to received it (i.e. not in the proper state).

Requirement 18

While not available nor idle, the host shall explicitely (i.e. with a dedicated message) deny any chat initiation requests.

Requirement 19

The reception of a chat initiation deny during a chat initiation procedure shall stop the ongoing procedure and the user shall be considered available again.

Chat sessions are modeled as two users sending each other chat data. A user can also choose to terminate the chat with a given probability. But it can also happen that a chat is terminated because the communication between the hosts is broken.

Requirement 20

Upon receiving chat data, a user shall reply with data or explicitely finish the chat with a given probability p and then be considered available again.

Requirement 21

The reception of a explicit finish message should terminate the current chat session and the user shall be considered available again.

Requirement 22

If no chat data is received for a certain time, the chat session should be considered terminated, and the user be considered available again.

For sake of example, the following figure shows some possible chat sessions:

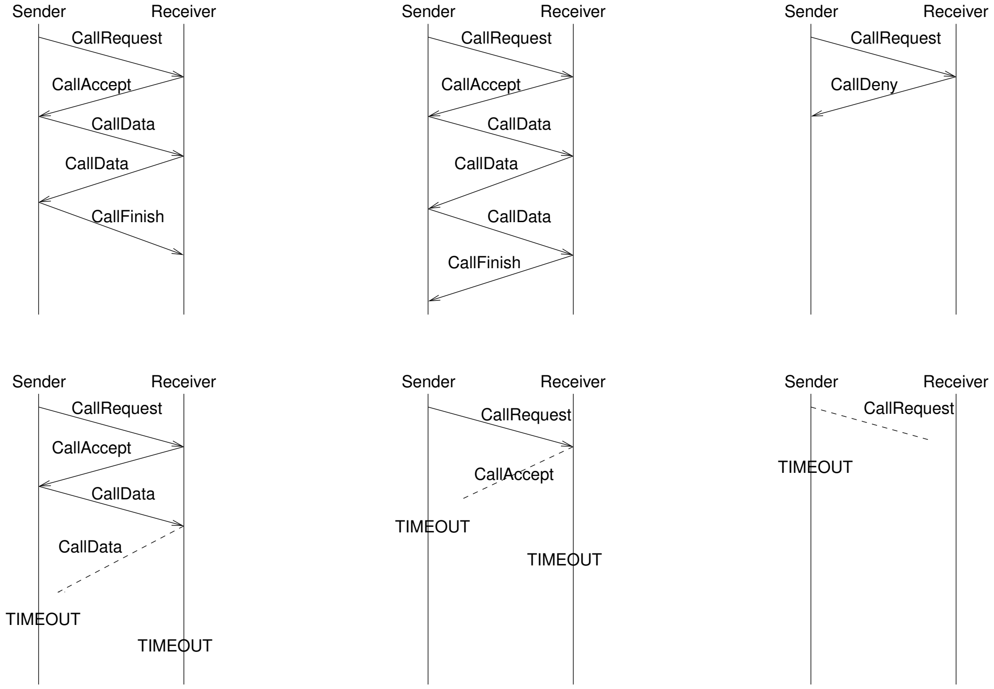

Figure 1: Chat sequences examples

3 Finite state machines and protocol modeling

3.1 Objectives

This section introduces the concept of finite state machine, and presents the modeling work you will have to do so as to represent the instant messenging protocol for ad hoc networks as a FSM.

After this step, you should have a state diagram representation of your protocol matching the different requirements and be ready for implementation.

3.2 Introduction to finite state machines

A finite state machine is a model of behavior composed of a finite number of states, transitions between those states, and actions. A finite state machine is an abstract model of a machine with a primitive internal memory.

A current state is determined by past states of the system. As such, it can be said to record information about the past, i.e. it reflects the input changes from the system start to the present moment.

A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition.

An action is a description of an activity that is to be performed at a given moment. There are several action types:

- **Entry action**, which is performed when entering the state;

- **Exit action**, which is performed when exiting the state;
- **Input action**, which is performed depending on present state and input conditions;
- **Transition action**, which is performed when performing a certain transition.

An FSM can be represented using a state diagram (or state transition diagram) or by several state transition table (like the event response table).

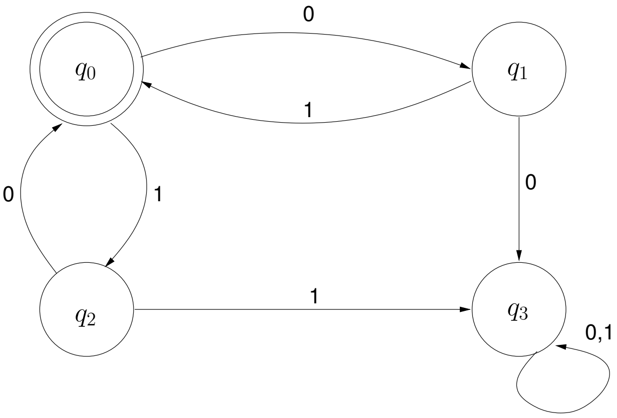

Figure 2: An example of finite state machine as state diagram

In networking, finite state machines are a convenient way to represent how a protocol should behave, as often protocol can be described as a set of states and follow a deterministic evolution. In this case, transition are most of the time the reception of a message or the expiration of a timer. Actions include generating and sending messages, scheduling timers, updating a data set, etc.

A good example of a quite complex finite-state machine is the TCP connection management presented in figure 3.

3.3 Application to protocol modeling

Now that you have been given the basic requirements of our protocol and had an introduction to finite state machines, the natural question is how to design a FSM that will match the protocol description.

The problem is that, despite the fact that some state may be obvious in the description, we do not know all possible states of the system at this point of the design process.

Therefore, a possible approach is to start with the initial state and *walk* through the protocol model using a **Event Response Table**, as the one presented below.

State	Event	Condition	Action	Final state
`INIT`	...	...	...	...

Table 1: An example of Event Response Table

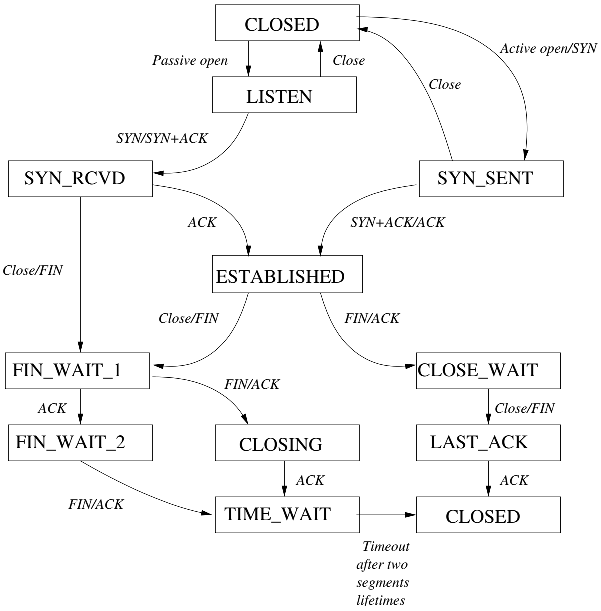

Figure 3: The TCP finite state machine as state diagram

Beginning from the `INIT` state, you can then use the following iteration plan:

Listing 1: State discovery iteration plan

```
Step 1: Choose a state
   Step 2: Chose an event
      Step 3: Chose a condition under which the event occurs
         Step 4: Determine all actions to perform
         Step 5: Determine the final state (may be a new one)
      Loop Step 3 for all conditions
   Loop Step 2 for all events
Loop Step 1 until all states are complete
(all states discovered during Step 5)
```

3.4 Modeling the instant messenging application protocol

Use the Event Response Table and the iteration plan so as to model the instant messenging application protocol as a finite state machine.

Once you think you have explored all the possible states (don't limit yourself to the obvious states described in 2.1.2), make a state diagram for your design presenting the different states, transition conditions and transition actions. You can also add the timers associated to each defined state.

You should also identify all the messages required for the protocol to perform, and organize them in classes of messages.

The state diagram creation is part of your lab project. It should be turned in at the end of lab 03 so as to be checked against the set of requirements and the protocol implementation you are about to perform.

4 Protocol implementation

4.1 Objectives

In this step, you will implement in OMNeT++ the finite state machine you have designed in the previous section.

You will be provided with some advice on implementation and given some code snippets to help you.

You can also have a look at the extensive documentation that comes with OMNeT++ and Mobility Framework :

- OMNeT++ manual
- OMNeT++ API documentation
- TicToc tutorial (which served as basis for the first lab)
- Mobility Framework manual
- Mobility Framework API documentation

Besides, note that OMNeT++ and Mobility Framework provide a lot of implemented modules and templates for you to start with. These files are excellent examples for how to implement more complicated modules. See for instance `omnetpp-3.4b2/samples`, `omnetpp-3.4b2/mobility-fw2.0p3/networks` or `omnetpp-3.4b2/mobility-fw2.0p3/templates`

This implementation will also make you use more intensively C++ and its standard library. A good documentation in point is available at `http://www.cppreference.com/`.

Finally, this document suggests an iterative approach for implementation, it is probably safer for you to follow it rather than implementing the protocol as a whole in one time.

4.2 Getting started

4.2.1 Setting the project directory

Start with the wireless network modeled in previous lab. It is a wise decision to copy all files to a new directory called `lab02` and use this one as new project directory.

Run the following commands in a Terminal window:

```
cd /home/user/Development
mkdir lab02
cp -r lab01 lab02
cd lab02
make clean
```

Now you should be ready to start.

4.2.2 Creating an application layer for our protocol

In the previous lab, you learned how to subclass a module (remember that you subclassed `Flood` and `BurstApplLayer` in order to add statisctics collection).

In this lab, you will subclass the most generic module for application layer, called `BasicApplLayer` in oder to implement a fully-functionnal module for the instant messenging protocol you have designed. Call this module `CallApplLayer` as, in all the remaining part of this document we will refer to a chat session as a *call*.

Instead of starting from a blank page, you can use the `YourApplLayer` module template located in the `/home/user/Develoment/omnetpp-3.4b2/mobility-fw2.0p3/template` directory.

You can also use the `CallApplLayer` module files located in the `/home/user/Development/template/lab02` directory. It contains the basic required for you to implement the module. Using this template is easier but you will be more constrained as you will have to follow the pattern of functions already defined.

Note that you will aslo have to update the `Host.ned` and eventually the `Network.ned` files for your host model to take into account the `CallApplLayer` you have designed.

4.2.3 Implementation advice

Here are some indications for you to be more efficient in implementing the instant messenging protocol:

- Use **message kind**. While creating a new message, use the constructor with two arguments : message name (string) and message kind (integer or item from an `enum`).
- Use **message kind** to distinguish between message of the same class. This is particularly easy using the C++ `switch` statement.
- You can also use the `switch` statement on states.
- Use generic `cMessage` class for timers, use **kind** to distinguish between different timers.
- Use `scheduleAt(time, msg_ptr)` to schedule a timer. At simulation time `time`, you will have to handle the message in `handleSelfMsg()`.
- Current simulation time can be obtained by `simTime()`. You can add a value of type `simtime_t` (which is a `typedef` of `double`) to this value to schedule a message in the future.
- Do not put everything in the `handleSelfMsg()` and `handleLowerMsg()` functions. Just separate on message kind here. Then call a dedicated `handleKindMsg()` function to do the dedicated processing. This is cleaner.
- Create a dedicated `sendKind()` function per message kind.
- For random numbers, use the random-number generators provided by OMNeT++ . See the documentation for further details.
- Use the `EV << "My output" << endl;` feature to generate traces for your simulation. This will help you understand what is happening.

- Use some GUI updates and enclose them in `if(ev.isGUI()) {}` condition blocks. So you will follow protocol behavior while in graphical mode (tkenv) but those statements won't be evaluated while in command-line mode (cmdenv).

4.3 Users discovery

To implement users discovery you will need the following:

- A beacon packet type;
- A neighborhood class so you can create a vector of neighbors to store the known users;
- Functions to periodically send your beacon, parse received messages and check the validity of your known neighbors entries.

4.3.1 Beacon messages

Implement a `BeaconPkt` message type (a MSG file) extending the `ApplPkt` message type that is used by the Mobility Framework . Those messages should contain the following fields:

- `state`, the current state of the node
- `timestamp`, the simulation time at which the message was generated.

Extending a message type is explained in the Mobility Framework documentation.

Once finished, don't forget to compile your message type using `opp_msgc`.

4.3.2 Neighborhood management

For neighborhood storage, I recommand using one of the C++ standard library (STL) containers (such as list, vector, or map). The template provided uses a map of pointers to `Neighbors` objects, indexed by the application layer address (witch is also their index as hosts). But this container is, by far, not the quickest depending on the operations you want to perform (accessing the i^{th} element, etc.). So if you are comfortable with STL, you can go and implement neighborhood management as you like.

For this feature to work properly, you need to implement :

- a function periodically sending the host current state in beacon messages;
- a function parsing all received beacon messages and keeping the list of known users up-to date :
  - create a new entry;
  - update an existing entry;
  - do nothing (an older message can arrive out-of sync and provide outdated information).
- a function deleting old entries. A proposed approach is that entries have an expiration time, and all expired entries should be removed.

4.4 State management

So as to respect an iterative approach for implementation, we will now focus on requirements described in section 2.1.2.

This can be done either by using the OMNeT++ class for finite state machines (refer to the OMNeT++ documentation for furher help on this) or simply by using an `enum` structure to define the states, store the current state in a variable and using a dedicated function to switch state. The latter approach is used in the provided template.

In the state switching function, you should implement:

- **exit actions** (i.e. what to perform before leaving the current state);
- **entry actions** (i.e. what to perform while entering the new state);
- **GUI update** so we can easily follow in what state is each host

Once this state switching function is implemented, it should be called when an occuring event is a condition for transitioning from the current state to a new state (according to the FSM and state diagram you have designed).

Such events can be:

- expiration of a timer (i.e. self-message arrival);
- arrival of a message (i.e. a lower-message to handle);
- sending a new message;
- particular value of a host attribute (e.g. the list of known users).

As we do not consider call (chat session) management here, the only external messages you will have to handle so far are beacons received from other nodes.

4.5 Call management

We will now focus on protocol requirements from section 2.1.3, related to call management.

4.5.1 Call messages

Implement a `CallPkt` message type (a MSG file) extending the `ApplPkt` message type that is used by the Mobility Framework . Those messages should contain the following fields:

- `timestamp`, the simulation time at which the message was generated.

This message type will be used for all call session related operations. You can use the `kind` attribute to distinguish between operations.

Extending a message type is explained in the Mobility Framework documentation.

Once finished, don't forget to compile your message type using `opp_msgc`.

4.5.2 Call sessions management

In your finite state machine design, call management operations should have introduced one or several new states and associated transitions.

Adding call sessions management to your protocol implementation should then be limited to:

- extending the state transition function to handle those states;
- adding new timers and the associated processing when they expire;
- extending the function handling lower messages as messages other than beacon can now be received from other hosts;
- creating additional functions to send messages as a node is now likely to send messages other than beacon.

4.6 Simulation configuration, build and run

Configure your simulation by udpating you `omnetpp.ini` file. The best way is to include a dedicated `call-protocol.ini` file with the parameters associated to your protocol.

Build the simulation (`opp_mf_makemake`, `make clean`, `make`) and launch it. You should see your nodes changing state (and thus, color).

4.7 Debugging

By inspecting the simulation traces (those you have created using `EV`), ensure that the protocol behavior is correct (having the simulation not crashing is not enough to ensure proper implementation of the protocol...).

You can use the GUI function that will dump the content of the log window to a file, and then parse this file using `grep` for instance (type `man grep` in a terminal window if you are not familiar with this tool).

You can also compile the simulation to run in cmdenv (`opp_mf_makemake_cmd`, `make clean`, `make`), launch the simulation while redirecting the output to a log file. Use for instance `./lab02 > log.txt`.

Note that in cmdenv, the simuation runs much faster (especially when the output is written to a file, not to the console). So if you use this method, don't forget to interrupt the simulation, by pressing `CTRL + C`, before the log file goes too large (a few seconds is enough). We will learn how to limit the simulation time in next lab.

Next time:

In next lab we will add statistics collection to our protocol model in order to assess the protocol performance. Then we will learn how to use more efficiently cmdenv and `omnetpp.ini` to automatically launch different simulation runs with various parameters, various seeds for random numbers (allowing some statistical confidence in our results), etc.