Stepwise Development of Peer-to-Peer Systems

Lu Yan

Turku Centre for Computer Science (TUCS) and
Department of Computer Science, Åbo Akademi University,
FIN-20520 Turku, Finland.

Kaisa Sere

Department of Computer Science, Åbo Akademi University and
Turku Centre for Computer Science (TUCS),
FIN-20520 Turku, Finland.

Turku Centre for Computer Science
TUCS Technical Report No 522
March 2003

ISBN 952-12-1153-9
ISSN 1239-1891

Abstract

Peer-to-peer systems like Napster, Gnutella and Kazaa have recently become popular for sharing information. In this paper, we show how to design peer-to-peer systems within the action systems framework by combining UML diagrams. We present our approach via a case study of stepwise development of a Gnutella-like peer-to-peer system.

Keywords: Peer-to-peer, action systems, UML, specification, Gnutella.

TUCS Laboratory
Programming Methodology Research Group

1 Introduction

Peer-to-peer systems like Napster, Gnutella and Kazaa have recently become popular for sharing information. People find in peer-to-peer applications a convenient solution to exchange resources via internet. Two factors have fostered the recent explosive growth of such systems: first, the low cost and high availablity of large numbers of computing and storage resource, and second, increased network connectivity. As these trends continue, the peer-to-peer paradigm is bound to be more popular[9].

Most current distributed systems like the Web follow the client-server paradigm in which a single server stores information and distributes it to clients upon their requests. Peer-to-peer systems, which consider that all nodes are equal for sharing information, on the other hand, follow a paradigm in which each node can store information of its own and establish direct connections with another node to download information. In this way, the peer-to-peer systems offer attractive advantages like enhanced load balancing, dynamic information repositories, redundancy and fault tolerance, content-based addressing and improved searching[11] over the traditional client-server systems.

Because of the surprisingly rapid deployment of some peer-to-peer applications and the great advantages of the peer-to-peer paradigm, we are motivated to conduct a study of peer-to-peer systems and achieve a way to develop such systems. After using and analyzing various peer-to-peer clients on different platforms, we identified two common problems of clients: *reliability and robustness* (Most clients fail to provide satisfactory download service and some buggy ones even bring down the system during our test) and *extendability* (Most clients are implemented in a way that adding new services or functionalities results to lots of modifications to the original specifications). An attractive strategy to solve the first open problem is to use formal methods in designing peer-to-peer systems. Formal methods can help with reliability and robustness by minimizing errors in designing peer-to-peer systems. To improve extendability, we introduce a modular and object-oriented architecture for peer-to-peer systems. The benefit of object-orientation can be used to design and implement peer-to-peer systems in a reusable, composable and extendable way.

In this paper, we show how to design peer-to-peer systems within the action systems framework by combining UML diagrams. We present our approach via a case study of stepwise development of a Gnutella-like peer-to-peer system. We start by briefly describing the action systems framework to the required extent in Section 2. In Section 3 we give an initial specification of the Gnutella system. An abstract action system specification is derived in Section 4. In Section 5 we analyze and refine the system specification introduced in the previous section. We end in Section 6 with concluding remarks.

2 Action systems

Action systems have proved to be very suitable for designing distributed systems[1, 2, 3, 6, 13]. The design and reasoning about action systems are carried out with refinement calculus[12].

In this section we will introduce OO-action systems[4], an object-oriented extension of action systems which we select as a foundation to work on. In this way, we can address our two open problems in a unified framework with benefits from both formal methods and object-orientation.

An OO-action system consists of a finite set of classes, each class specifying the behaviour of objects that are dynamically created and executed in parallel.

2.1 Actions

We will consider a fixed set *Attr* of attributes (variables) and assume that each attribute is associated with a nonempty set of *values*. Also, we consider a set *Act* of actions defined by the following grammar

$$A \ ::= \ abort|skip|x := v|x :\in V|b \rightarrow A|\{b\}|A;A|A \ [] \ A.$$

Here x is a list of attributes, v a list of values, V a nonempty set of values, b a predicate over attributes. Intuitively, $abort$ is the action which always deadlocks, $skip$ is a stuttering action, $x := v$ is a multiple assignment, $x :\in V$ is a random assignment, $b \rightarrow A$ is a guard of an action, $\{b\}$ is an assertion, $A_1; A_2$ is the sequential composition of the actions A_1 and A_2, and $A_1 \, [] \, A_2$ is the nondeterministic choice between the action A_1 and A_2.

The *guard* of an action is defined in a standard way using weakest preconditions[14]

$$g(A) \ = \ \neg wp(A, \textit{false}).$$

The action A is said to be enabled when the guard is true.

2.2 Classes and objects

Let *CName* be a fixed set of class names and *OName* a set of valid names for objects. We will also consider a fixed set of object variables *OVar* assumed to be disjoint from *Attr*. The only valid values for object variables are the names for objects in *OName*. The set of object actions *OAct* is defined by extending the grammar of actions as follows

$$\begin{aligned} O \quad ::= \quad & A|n := o|new(c)|n := new(c) \\ & |p|n.m|\textbf{\textit{self.m}}|super.m|O;O|O \ [] \ O. \end{aligned}$$

Here $A \in Act$, n is an object variable, o is either an object name or the constants *self* or $super$ (all three possibly resulting from the evaluation of an expression), c is a class name, p a procedure name, and m is a method name. Intuitively, $n := o$ stores the object name o into the object variable n, $new(c)$ creates a new object instance of the class c, $n := new(c)$ assigns the name of the newly created instance of the class c to the object variable n, p is a procedure call, $n.m$ is a call of the method m of the object the name of which is stored in the object variable n, *self.m* is a call of the method m declared in the same object, and $super.m$ is a call of the method m declared in the object that created the calling object. Note that method calls are always prefixed by an object variable or by the constant *self* or $super$.

We define the guard $gd(O)$ of an object action O to be the guard of the action in Act obtained by substituting every atomic object action of O with the action $skip$, where an atomic object action is

$$A, n := o, new(c), n := new(c), p, n.m, \textit{self.m}, super.m.$$

The resulting statement is an action in Act and hence its guard is well defined.

A $class$ is a pair $< c, \mathcal{C} >$, where $c \in CName$ is the $name$ of the class and $\mathcal{C}$ is its $body$, that is, a statement of the form

$$\begin{array}{lll} \mathcal{C} \ = \ |[& \mathbf{attr} & y^* := y0; x := x0 \\ & \mathbf{obj} & n \\ & \mathbf{meth} & m_1 = M_1; \cdots ; m_h = M_h \\ & \mathbf{proc} & p_1 = P_1; \cdots ; p_k = P_k \\ & \mathbf{do}\ O\ \mathbf{od} & \\ ]| & & \end{array}$$

A class body consists of an object action O and of four declaration sections. In the attribute declaration the *shared attributes* in the list y, marked with an asterisk $*$, describe the variables to be shared among all active objects. Therefore they can be used by instances of the class $\mathcal{C}$ and also by object instances of other classes. Initially they get values $y0$. The *local attributes* in the list x describe variables that are local to an object instance of the class, meaning that they can only be used by the instance itself. The variables are initialized to the value $x0$.

The list n of *object variables* describes a special kind of variables local to an object instance of the class. They contain names of objects and are used for calling methods of other objects. We assume that the lists x, y and n are pairwise disjoint.

A *method* $m_i = M_i$ describes a procedure of an object instance of the class. They can be called by actions of the objects themselves or by actions of another object instance of possibly another class. A method consists of a method name m and an object action M.

A *procedure* $p_i = P_i$ describes a procedure that is local to the object instances of the class. It can be called only by actions of the object itself. Like a method, it consists of a procedure name p and an object action forming the body P.

The *class body* is a description of the actions to be executed repeatedly when the object instance of the class is activated. It can refer to attributes which are declared to be shared in another class, and to the object variables and the local attributes declared within the class itself. It can contain procedure calls only to procedures declared in the class and method calls of the form $n.m$ or $super.m$ to methods declared in other classes. Method calls *self.m* are allowed only if m is a method declared in the same class. As for action systems, the execution of an object action is atomic.

2.3 OO-action system

An *OO-action system* OO consists of a finite set of classes

$$OO \ = \ \{< c_1, \mathcal{C}_1 >, \cdots, < c_n, \mathcal{C}_n >\}$$

such that the shared attributes declared in each $\mathcal{C}_i$ are distinct and actions in each $\mathcal{C}_i$ or bodies of methods and procedures declared in each $\mathcal{C}_i$ do not contain new statements referring to class names not used by classes in OO. Local attributes, object variables, methods, and procedures are not required to be distinct.

There are some classes in OO, marked with an asterisk $*$. Execution starts by the creation of one object instance of each of these classes. Each object, when created, chooses enabled actions and executes them. Actions operating on disjoint sets of local and shared attributes, and object variables can be executed in parallel. They can also create other objects. Actions of different active objects can be executed in parallel if they are operating on disjoint sets of shared attributes. Objects interact by means of the shared attributes and by executing methods of other objects.

3 Initial specification of the Gnutella system

Gnutella is a decentralized peer-to-peer file-sharing model developed in 2000 by Nullsoft, AOL subsidiary and the company that created WinAMP [10]. The Gnutella model enables file sharing without using servers.

Unlike a centralized server network, the Gnutella network does not use a central server to keep track of all user files. To share files using the Gnutella model, a user starts with a networked computer A with a Gnutella *servent*, which works both as a server and a client. Computer A will connect to another Gnutella-networked computer B and then announce that it is *alive* to computer B. B will

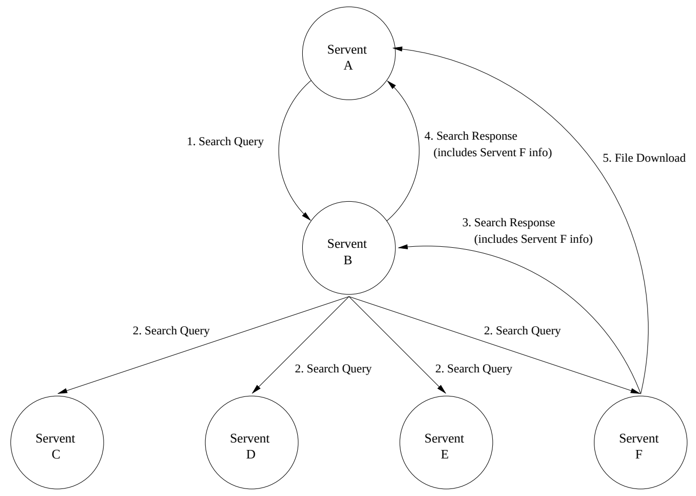

Figure 1: Gnutella peer-to-peer model[10]

in turn announce to all its neighbours C, D, E, and F that A is alive. Those computers will recursively continue this pattern and announce to their neighbours that computer A is alive. Once computer A has announced that it is alive to the rest of the members of the peer-to-peer network, it can then search the contents of the shared directories of the peer-to-peer network.

Search requests are transmitted over the Gnutella network in a decentralized manner. One computer sends a search request to its neighbours, which in turn pass that request along to their neighbours, and so on. Figure 1 illustrates this model. The search request from computer A will be transmitted to all members of the peer-to-peer network, starting with computer B, then to C, D, E, F, which will in turn send the request to their neighbours, and so forth. If one of the computers in the peer-to-peer network, for example, computer F, has a match, it transmits the file information (name, location, etc.) back through all the computers in the pathway towards A (via computer B in this case). Computer A will then be able to open a direct connection with computer F and will be able to download that file directly from computer F.

4 Action system specification of the Gnutella system

When taking a step back, it is seen that the Gnutella system enables at least the following functionalities:

1. Servent can easily join and leave the peer-to-peer network.

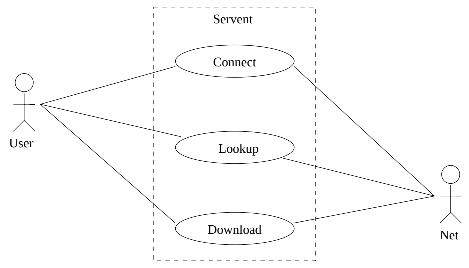

Figure 2: Use Case diagram of servent

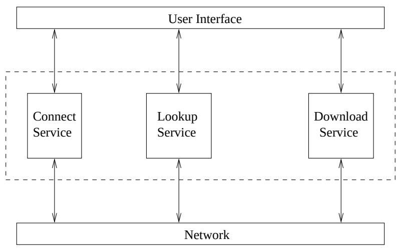

Figure 3: Structure diagram of servent

2. Servent can publish its content to the shared directories of the peer-to-peer network.

3. Servent can search for and download files from the shared directories of the peer-to-peer network using keywords.

Based on the simple descriptions above, we can identify that servent should provide three basic services, *connect service*, *lookup service* and *download service*, as shown in Fig.2. From this diagram we divide the system into components and derive a component-based structure of servent in Fig.3.

The statechart diagram Fig.4 shows the joint behaviour of servent. Each state is described by a set of attributes. We give unique preconditions for entering each state. Table 1 shows preconditions and invariants for every state of the servent.

An interesting issue to notice is that downloads can be initiated in two ways, i.e. either from a search result or by directly specifying target information. This design is reasonable because we do not always need to search the peer-to-peer network to get wanted files. In some cases, name and location information of files is already available. For example, file exchanges between two friends, who have already known each other's IP and shared contents. Take this into consideration, we provide two ways to initiate downloads.

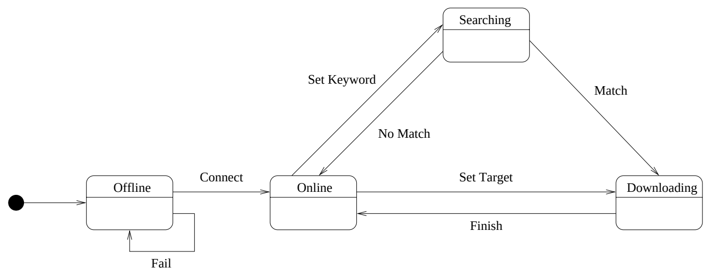

Figure 4: Statechart diagram of servent

Table 1: Preconditions and invariants for states

State	Precondition	Invariant
Offline	$\neg$connected $\wedge$ keyword = ϕ $\wedge$ target = ϕ	keyword = ϕ $\wedge$ target = ϕ
Online	connected $\wedge$ keyword = ϕ $\wedge$ target = ϕ	connected
Searching	connected $\wedge$ keyword $\neq \phi$ $\wedge$ target = ϕ	connected $\wedge$ target = ϕ
Downloading	connected $\wedge$ keyword = ϕ $\wedge$ target $\neq \phi$	connected $\wedge$ keyword = ϕ

Table 2: Initial specification of servent

```
GS = |[ attr  $connected^*; keyword^*; target^*; state := \mathit{Offline}$
        do
              $state = \mathit{Offline} \wedge connected \rightarrow$
                  $state := Online$
              $[]\ state = Online \wedge keyword \neq \phi \rightarrow$
                  $state := Searching$
              $[]\ state = Online \wedge target \neq \phi \rightarrow$
                  $state := Downloading$
              $[]\ state = Searching \wedge target = \phi \rightarrow$
                  $keyword := \phi; state := Online$
              $[]\ state = Searching \wedge target \neq \phi \rightarrow$
                  $keyword := \phi; state := Downloading$
              $[]\ state = Downloading \rightarrow$
                  $target := \phi; state := Online$
        od
     ]|
```

The first version of action system specification of servent can be derived directly from Fig.4 and Table 1.

$$\{< GnutellaServent, GS >, \cdots\}$$

where the class body in Table 2 consists of attribute declaration, initialisation and a loop of actions which are chosen for execution in a non-deterministic fashion when enabled. Each action is of the form $g \rightarrow S$ where g is the guard and S is a statement to be executed when the guard evaluates to $true$. Here *connected* is a boolean variable; *keyword* is the search criteria; *target* is the location information of shared resources in format *filename@IP*.

The next step is to apply the design in Fig.3 to our initial specification, which results in three more classes *ConnectService, LookupService* and *DownloadService*. Now the system consists of a set of classes $< c, C >$ where c is the name of the class and C is its body. On the top level, we have components of servent

$$\{< GnutellaServent, GS >^*, < ConnectService, Cs >,$$
$$< LookupService, Ls >, < DownloadService, Ds >, \cdots\}$$

The class $< GnutellaServent, GS >$, marked with an asterisk $*$, is the root class. At execution time one object of this class is initially created and this in turn creates other objects by executing new statements.

Table 3: Refined specification of servent

```
GS = |[ attr  $connected := \mathit{false}; keyword := \phi; target := \phi$
        obj   $c : ConnectService; l : LookupService;$
              $d : DownloadService$
        meth  $SetKeyword(k) = keyword := k;$
              $SetTarget(t) = target := t$
        init  $c := new(ConnectService);$
              $l := new(LookupService);$
              $d := new(DownloadService)$
        do
               $\neg connected \rightarrow connected := c.Connect(\ )$
              $[]$ $connected \wedge keyword \neq \phi \rightarrow$
                   $target := l.Search(keyword); keyword := \phi$
              $[]$ $connected \wedge target \neq \phi \rightarrow$
                   $d.Download(target); target := \phi$
        od
     ]|
```

Let us look at the actions in Table 2. We can refine them according to service groups. For example, action

$$state = \mathit{Offline} \wedge connected \rightarrow state := Online$$

where *Offline* and *Online* are defined in Table 1, can be refined into action

$$\neg connected \rightarrow connected := c.Connect(\ )$$

In this paper, however, we skip refinement details here because we do not want to go into details of semantics of action systems[15] nor refinement rules of refinement calculus[12].

The body of the refined specification of servent is described in Table 3. It models a servent that provides three basic services (*ConnectService, LookupService* and *DownloadService*). When it connects itself to the peer-to-peer network, users can search the network via $SetKeyword$ method and then download files from the search result. Or alternatively, users can directly give *target* information via $SetTarget$ method to download files.

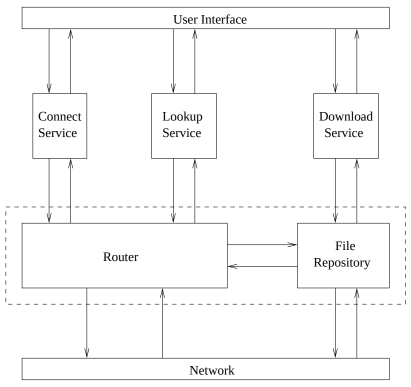

Figure 5: Schematic diagram of servent

5 Refining Gnutella servent

Ultimately, we need to derive an implementable specification for each service in the Guntella servent. Hence, every service component should be refined. We notice *ConnectService* and *LookupService* share a common functionality that enables appropriate message routing. It is reasonable to introduce a new component *Router* to the system as depicted in Fig. 5. This component will be in charge of routing all the incoming and outgoing messages of the servent. For *DownloadService*, we introduce a new component *FileRepository* in Fig. 5. It will act as a resource exchanger between servent and network.

5.1 Refining *ConnectService*

We start by considering *ConnectService* first. A Gnutella servent connects itself to the peer-to-peer network by establishing a connection with another servent currently on the network, and this kind of connection is passed around recursively. In order to model the communication between servents, we define a set of descriptors and inter-servent descriptor exchange rules as follows[7]

Ping Used to actively discover hosts on the network. A servent receiving a Ping descriptor is expected to respond with one or more Pong descriptors.

Pong The response to a Ping. Includes the address of a connected Gnutella servent and information regarding the amount of data it is making available to the network.

Furthermore, we need to define the format of Ping descriptor and Pong descriptor. We use the message format in Table 4, where *DescriptorID* is a string

Table 4: Message format

$$
\begin{array}{lll}
Msg = |[& \mathbf{attr} & descriptorID; TTL; type; \mathit{info} \\
 & \mathbf{meth} & Transmit(\,) = TTL > 0 \to TTL := TTL - 1 \\
 & \mathbf{init} & t \in \{Ping, Pong\} \to \\
 & & \quad Msg(t) = (descriptorID := \_unique\_ID\_; \\
 & & \quad TTL := \_max\_TTL\_; type := t; \mathit{info} := \_\mathit{info}\_) \\
 ]| & &
\end{array}
$$

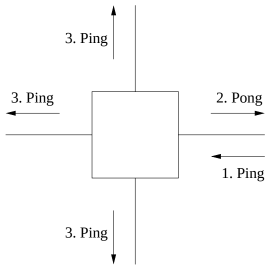

Figure 6: Ping - Pong routing[7]

uniquely identifying the descriptors on the network. TTL stands for *Time To Live*, which is the number of times the descriptor will be forwarded by servent before it is removed from the network. The TTL is the only mechanism for expiring descriptors on the network. Each servent will decrement the TTL before passing it on to another servent. When the TTL reaches 0, the descriptor will no longer be forwarded. The information carried by the message is encoded in *info*, whose format depends on the variable *type*.

The peer-to-peer nature of the Gnutella network requires servents to route network traffic appropriately. Intuitively, a servent will forward incoming Ping descriptors to all its directly connected servents. Pong descriptors may only be sent along the same path that carried the incoming Ping descriptor as shown in Fig.6. This ensures that only servents that routed the Ping descriptors will see the Pong descriptor in response. A servent that receives a Pong descriptor with $descriptorID = n$, but has not seen a Ping descriptor with $descriptorID = n$ should remove the Pong descriptor from the network.

The above routing rules can be illustrated in the statechart diagram Fig. 7. Using the same techniques as in the previous section, we can translate the diagram into action system specification and further refine it into Table 5.

The specification of Ping - Pong router models a router that can route Ping - Pong traffic appropriately. When the router is initiating, it connects itself to

Table 5: Specification of Ping - Pong router

```
$Rc = |[$  attr  $connected := false; descriptorDB := \phi; peers := \phi$
           obj   $receivedMsg : Msg; newMsg : Msg$
           meth  $ReceiveMsg(\ ) = receivedMsg := \_incoming\_message\_;$
                 $SendPing(\ ) = (newMsg := new(Msg(Ping));$
                       $\_outgoing\_message\_ := newMsg);$
                 $SendPong(\ ) = (newMsg := new(Msg(Pong));$
                       $newMsg.info.IP := \_this\_IP\_;$
                       $\_outgoing\_message\_ := newMsg);$
                 $ForwardMsg(m) = (m.TTL > 0 \rightarrow$
                       $m.Transmit(\ ); \_outgoing\_message\_ := m)$
           init  $SendPing(\ )$
           do
                  $|[\ peers \neq \phi \rightarrow connected := true$
                  $[]\ \neg connected \rightarrow SendPing(\ )\ ]|$
                  $/\!/$
                  $|[\ true \rightarrow$
                        $ReceiveMsg(\ );$
                        if $receivedMsg.type = Ping \rightarrow$
                              $descriptorDB := descriptorDB \cup$
                              $receivedMsg.descriptorID;$
                              $SendPong(\ );$
                              $ForwardMsg(receivedMsg)$
                        $[]\ receivedMsg.type = Pong \rightarrow$
                              $peers := peers \cup receivedMsg.info.IP;$
                              $receivedMsg.descriptorID \in descriptorDB \rightarrow$
                                    $ForwardMsg(receivedMsg)$
                        fi
                  $]|$
           od
       $]|$
```

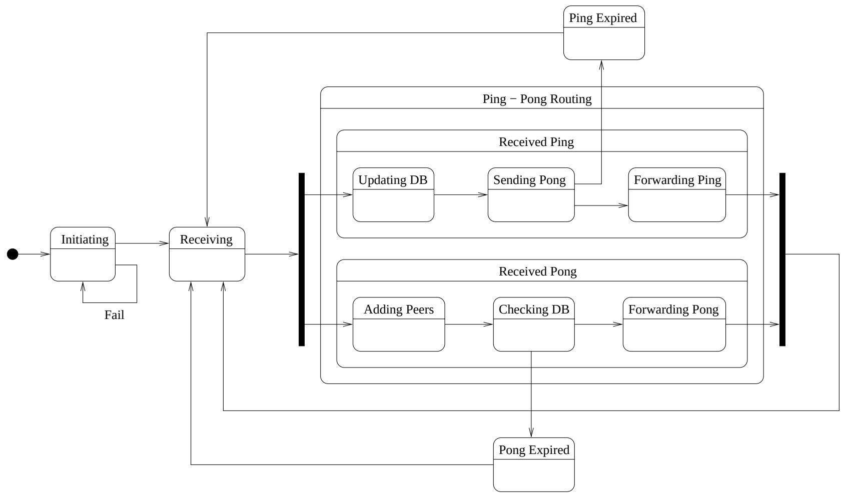

Figure 7: Statechart diagram of Ping - Pong routing rules

the peer-to-peer network by sending Ping descriptors to other peers. After initiation, it continues receiving *_incoming_message_* and replying with apporiate *_outgoing_message_*. Here *descriptorDB* is a set storing *descriptorID* information; *peers* is a set storing its directly and indirectly connected servents information; and *_this_IP_* is the IP address of the responding servent. The sequence of a connect session is summarized in Fig. 8.

In order to reuse the specification in Table 3, we will specify *ConnectService* without making any changes in its interface. The specification is shown in Table 6. When *ConnectService* is initiating, an instance of *Router* is created. Then it keeps checking state variable *connected* in the router and passing the status to servent.

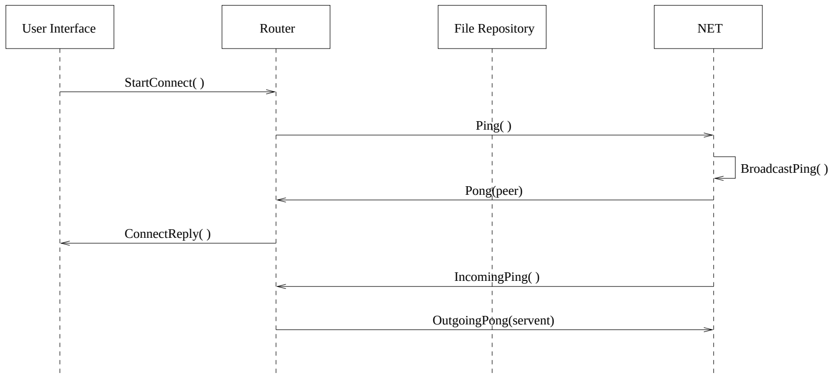

Figure 8: Sequence diagram of a connect session

Table 6: Specification of *ConnectService*

$$
\begin{array}{rcll}
Cs & = & |[& \textbf{attr} \quad connected := \textit{false} \\
 & & & \textbf{obj} \quad r : Router \\
 & & & \textbf{meth} \quad Connect(\,) = (connected := r.connected) \\
 & & & \textbf{init} \quad r := new(Router) \\
 & &]| &
\end{array}
$$

Table 7: Refined message format

$$
\begin{array}{rcll}
Msg & = & |[& \textbf{attr} \quad descriptorID; TTL; type; \textit{info} \\
 & & & \textbf{meth} \quad Transmit(\,) = TTL > 0 \rightarrow TTL := TTL - 1 \\
 & & & \textbf{init} \quad t \in \{Ping, Pong, Query, QueryHit\} \rightarrow \\
 & & & \qquad Msg(t) = (descriptorID := \_unique\_ID\_; \\
 & & & \qquad TTL := \_max\_TTL\_; type := t; \textit{info} := \_\textit{info}\_) \\
 & &]| &
\end{array}
$$

5.2 Refining *LookupService*

When we think about *LookupService*, we follow almost the same paradigm as *ConnectService* to specify this component. A Gnutella servent starts a search request by broadcasting a *Query* message through the peer-to-peer network. Upon receiving a search request, the servent checks if any local stored files match the query and sends a *QueryHit* message back. We use following descriptors and routing rules to model the communication between servents[7]

Query The primary mechanism for searching the distributed network. A servent receiving a Query descriptor will respond with a QueryHit if a match is found against its local data set.

QueryHit The response to a Query. This descriptor provides the recipient with enough information to acquire the data matching the corresponding Query.

The message format in Table 4 has to be revised to adopt the new descriptors. The message *type* now includes *Ping, Pong, Query* and *QueryHit*, so a minor change is made in Table 7.

The routing rules for Query - QueryHit traffic are similar to the rules for Ping - Pong traffic. A servent will forward incoming Query descriptors to all its directly connected servents. QueryHit descriptors may only be sent along the same

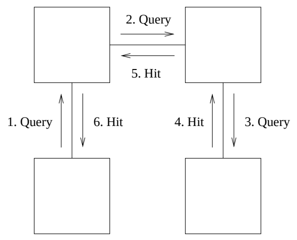

Figure 9: Query - QueryHit routing[7]

path that carried the incoming Query descriptor as shown in Fig. 9. This ensures that only those servents that routed the query descriptors will see the QueryHit descriptor in response. A servent that receives a QueryHit descriptor with $descriptorID = n$, but has not seen a Query descriptor with $descriptorID = n$ should remove the QueryHit descriptor from the network.

Like the previous section, we first draw a statechart diagram for the Query - QueryHit routing rules. Then we translate it into action system specification and further refine it.

In Table 8 we have the body of Query - QueryHit router specification, which models a router that is in charge of routing Query - QueryHit traffic appropriately. Like a Ping - Pong router, it keeps receiving *_incoming_message_* and replying with apporiate *_outgoing_message_*. Here *descriptorDB* is a set storing *descriptorID* information; *myKeyword* is a string storing search criteria; *cKeyword* is a string storing comparison criteria; *filename* is a string storing destination filename; *target* is the shared resource location information in format *filename@IP*; and *f* is an object of class *FileRepository* which enables local file search service via *Has* and *Find* methods. Details of class *FileRepository* will be elaborated in the next section.

The Query - QueryHit router provides searching function via method *SetKeyword*. Once a keyword is set, a Query descriptor carrying search criteria is generated and broadcasted in the peer-to-peer network via method *SendQuery*. In the mean time, the router keeps receiving Query and QueryHit descriptors. For an incoming Query descriptor, a query request is passed to *FileRepository*. According to the search result, a QueryHit descriptor is sent back in response via method *SendQueryHit* if a match is found, otherwise the Query descriptor is further forwarded via method *ForwardMsg*. Upon receiving a QueryHit descriptor, it checks its keyword field, and then sets *target* information to complete the search session. We summarize the sequence of a search session in Fig. 10.

Now we specify *LookupService* with emphasis on specification reuse. The result is shown in Table 9. When *LookupService* is initiated, an instance of *Router* is created. It provides *Search* method via calling *SetKeyword* method in the router,

Table 8: Specification of Query - QueryHit router

```
$Rl$ = |[ attr  $descriptorDB := \phi; myKeyword := \phi;$
                $cKeyword := \phi; \textit{filename} := \phi; target := \phi$
          obj   $receivedMsg : Msg; newMsg : Msg; f : \textit{FileRepository}$
          meth  $SetKeyword(k) = myKeyword := k;$
                $ReceiveMsg(\,) = receivedMsg := \_incoming\_message\_;$
                $SendQuery(\,) = (newMsg := new(Msg(Query));$
                     $newMsg.\textit{info}.keyword := myKeyword;$
                     $cKeyword := myKeyword; target := \phi;$
                     $\_outgoing\_message\_ := newMsg);$
                $SendQueryHit(\,) = (newMsg := new(Msg(QueryHit));$
                     $newMsg.\textit{info}.keyword := receivedMsg.\textit{info}.keyword;$
                     $newMsg.\textit{info}.\textit{filename} := \textit{filename};$
                     $newMsg.\textit{info}.IP := \_this\_IP\_;$
                     $\_outgoing\_message\_ := newMsg);$
                $\textit{ForwardMsg}(m) = (m.TTL > 0 \rightarrow$
                     $m.Transmit(\,); \_outgoing\_message\_ := m)$
          do
                 |[ $myKeyword \neq \phi \rightarrow SendQuery(\,); myKeyword = \phi$ ]|
                 //
                 |[ $true \rightarrow$
                      $ReceiveMsg(\,);$
                      if $receivedMsg.type = Query \rightarrow$
                           $descriptorDB := descriptorDB \cup$
                           $receivedMsg.descriptorID;$
                           if $f.Has(receivedMsg.\textit{info}.keyword) \rightarrow$
                                $\textit{filename} := f.Find(receivedMsg.\textit{info}.keyword);$
                                $SendQueryHit(\,)$
                           [] $\neg f.Has(receivedMsg.\textit{info}.keyword) \rightarrow$
                                $\textit{ForwardMsg}(receivedMsg)$
                           fi
                      [] $receivedMsg.type = QueryHit \rightarrow$
                           if $receivedMsg.\textit{info}.keyword = cKeyword \rightarrow$
                                $target := receivedMsg.\textit{info}.\textit{filename}@$
                                $receivedMsg.\textit{info}.IP; cKeyword := \phi$
                           [] $receivedMsg.\textit{info}.keyword \neq cKeyword \wedge$
                           $receivedMsg.descriptorID \in descriptorDB \rightarrow$
                                     $\textit{ForwardMsg}(receivedMsg)$
                           fi
                      fi
                 ]|
          od
         ]|
```

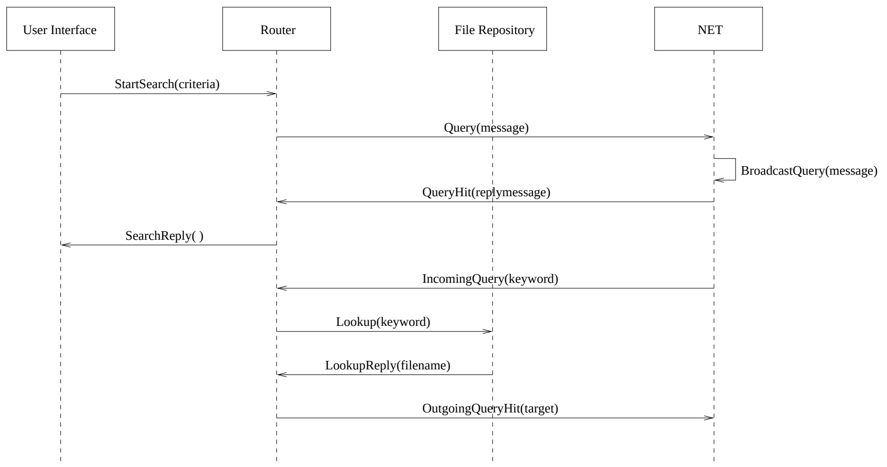

Figure 10: Sequence diagram of a search session

Table 9: Specification of *LookupService*

$$
\begin{array}{lll}
Ls \;=\; |[& \textbf{attr} & target := \phi \\
 & \textbf{obj} & r : Router \\
 & \textbf{meth} & Search(k) = (r.SetKeyword(k); target := r.target) \\
 & \textbf{init} & r := new(Router) \\
]| & &
\end{array}
$$

and then returning the search result to servent.

Until now we have two action systems, *Rc* modeling Ping - Pong routing rules and *Rl* modeling Query - QueryHit routing rules. We notice the two action systems actually model different aspects of a full router. Furthermore, we can compose the two action systems together using *prioritising composition*[5] to derive the action system specification of a full router

$$R \;=\; |[\; Rc \,/\!\!/\, Rl \;]|$$

where on the higher level, we have components of the router

$$\{< Router, R >, < PingPongRouter, Rc >, < QueryRouter, Rl >\}$$

5.3 Refining *DownloadService*

DownloadService is relatively simple compared to *ConnectService* and *LookupService*. The primary function of this component is to enable a servent to download

Table 10: Specification of file repository

```
F = |[ attr   fileDB := _fileDB_; filename := $\phi$; $target := \phi$
       meth   $SetTarget(t) = (target := t)$;
              $Has(key) = (\{key\} \in dom($fileDB$))$;
              $Find(key) = ($filename := file $\wedge \{$file$\} \in ran(\{key\} \lhd$ fileDB$))$
       do
               $target \neq \phi \rightarrow$
                     $HTTP(target)$;
                     $target := \phi$;
                     Refresh(fileDB)
       od
    ]|
```

files from other servents. Once a servent receives a QueryHit descriptor, it may initiate the direct download of one of the files described by the descriptor's result set. Or alternatively, users can initiate the download directly by giving complete *target* information. Files are downloaded out-of-network, i.e. a direct connection between the source and target servent is established in order to perform the data transfer. File data is never transferred over the peer-to-peer network.

Additionally, this component provides the local file query function for other servents. It should be in charge of a local file database which provides data services like *add*, *delete*, *update*, *query* and *refresh* etc. Moreover, it should take full control of local files. Hence we introduce a new component *FileRepository* in Table 10, which will satisfy the above requirements for *DownloadService*. First of all, we provide *SetTarget* method to enable file downloads. To make things simple, we assume that *fileDB* is simply a set of relations $\{key\} \rightarrow \{$*file*$\}$. We use relation notations[16] *dom* and *ran* for domain and range operations, and $\lhd$ as a domain restriction operator, defined by $S \lhd r = \{x, y | x \mapsto y \in r \wedge x \in S\}$. For incoming Query descriptors, *Has* and *Find* methods are provided to enable local file searches.

Given *target* information, download action is enabled and servent initiates a download. A download request is sent to the target servent, and then a file is downloaded via HTTP protocol. Afterwards, *fileDB* is refreshed in order to reflect the change of adding new files to the repository. The sequence of a download session is summarized in Fig. 11.

The last step is to specify *DownloadService*. From the result in Table 11, we can see that when *DownloadService* is initiating, an instance of *FileRepository* is created. It enables *Download* by calling *SetTarget* method in *FileRepository*.

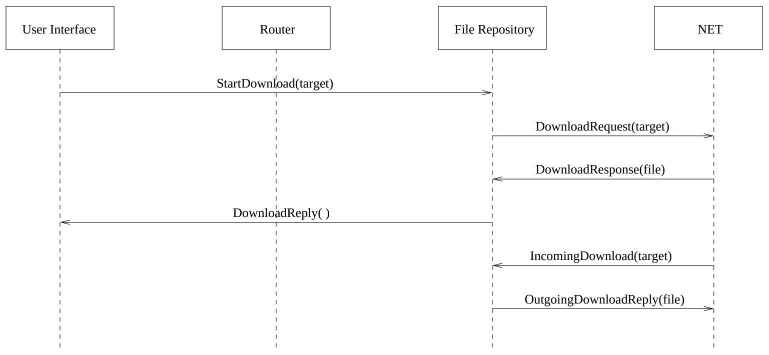

Figure 11: Sequence diagram of a download session

Table 11: Specification of *DownloadService*

$$
\begin{array}{lll}
Ds = |[& \mathbf{obj} & f : \textit{FileRepository} \\
 & \mathbf{meth} & Download(t) = f.SetKeyword(t) \\
 & \mathbf{init} & f := new(\textit{FileRepository}) \\
]| & &
\end{array}
$$

At this stage of the design, we finally have a complete set of classes which are refinement results from the initial specification of servent as follows

$$
\begin{gathered}
\{< GnutellaServent, GS >^*, < ConnectService, Cs >, \\
< LookupService, Ls >, < DownloadService, Ds >, \\
< PingPongRouter, Rc >, < QueryRouter, Rl >, \\
< Router, R >, < \textit{FileRepository}, F >, < Message, Msg >\}
\end{gathered}
$$

6 Concluding remarks

We identified two open problems of existing peer-to-peer systems: *reliability and robustness* and *extendability*, and proposed strategies that can be used to solve them. The main contribution of this paper is an approach to stepwise development of peer-to-peer systems within the action systems framework by combining UML diagrams. We have presented our approach via a case study of stepwise development of a Gnutella-like peer-to-peer system.

Our experience shows that it is beneficial to combine informal methods like UML and formal methods like action systems together in the development of peer-to-peer systems. In the early stage, we try to catch the characteristic of the system using use case diagrams and statechart diagrams. Then formal specification in action systems framework is derived by further studying and elaborating details of these diagrams. In the later stage, sequence diagrams are used to graphically clarify the structure of the refined action system specification.

Moreover, we find OO-action systems very suitable for designing such kind of systems. The formal nature of OO-action systems makes it a good tool to built reliable and robust systems. Meanwhile, the object-oriented aspect of OO-action systems helps to built systems in an extendable way, which will generally ease and accelerate the design and implementation of new services or functionalities. Furthermore, the final set of classes in the OO-action system specification is easy to be implemented in popular OO-languages like Java, C++ or C#.

Peer-to-peer systems have been evolving very quickly. Besides Gnutella, another promising choice is JXTA[8] from Sun, which has been generating lots of attention. In the future work, we plan to further investigate this new standard. Moreover, we plan to stepwise implement our action system specification and develop it into a real peer-to-peer system.

References

[1] R.J.R. Back and K. Sere: *From Action Systems to Modular Systems.* Software - Concepts and Tools. (1996) 17: 26–39.

[2] R.J.R. Back, A.J. Martin and K.Sere: *Specifying the Caltech asynchronous microprocessor.* Science of Computer Programming. (1996) 26: 79–97.

[3] L. Petre, M. Qvist and K. Sere: *Distributed Object-Based Control Systems.* TUCS Technical Reports, No 241, February 1999.

[4] M. Bonsangue, J.N. Kok and K. Sere: *An approach to object-orientation in action systems.* Proceedings of Mathematics of Program Construction (MPC'98), Marstrand, Sweden, June 1998. Lecture Notes in Computer Science 1422. Springer Verlag.

[5] E. Sekerinski and K. Sere: *A Theory of Prioritising Composition.* TUCS Technical Reports, No 5, May 1996.

[6] M. Butler, E. Sekerinski and K. Sere: *An Action System Approach to the Steam Boiler Problem.* In Jean-Raymond Abrial, Egon Borger and Hans Langmaack, editors, Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. Lecture notes in Computer Science Vol. 1165. Springer-Verlag, 1996.

[7] Clip2 DSS: *Gnutella Protocol Specification v0.4.*
Online. http://www.clip2.com/GnutellaProtocol04.pdf.

[8] L. Gong: *JXTA: A network programming environment.* IEEE Internet Computing, 5(3): 88–95, May/June 2001.

[9] M. Ripeanu: *Peer-to-peer architecture case study: Gnutella network.* Technical Report TR-2001-26, University of Chicago, Department of Computer Science, July 2001.

[10] I. Ivkovic: *Improving Gnutella Protocol: Protocol Analysis and Research Proposals.* Technical report, LimeWire LLC, 2001.

[11] M. Parameswaran, A. Susarla and A.B. Whinston: *P2P networking: An information-sharing alternative.* IEEE Computer, 34(7): 31–38, July 2001.

[12] R.J. Back and J. Wright: *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science, Springer-Verlag, 1998.

[13] L. Petre and K. Sere: *Coordination Among Mobile Objects.* Proceeding of IFIP TC6/WG6 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), Florence, Italy, February 1999.

[14] E.W. Dijkstra: *A Discipline of Programming.* Prentice-Hall International, 1976.

[15] K. Sere: *Stepwise derivation of parallel algorithms.* Academic dissertation of Åbo Akademi Department of Computer Science, 1990.

[16] E. Sekerinski and K. Sere (Eds): *Program Development by Refinement: Case Studies Using the B Method.* Springer-Verlag, 1999.