A Configurable Overlay Network Architecture

Sergio Ariel Salinas[1], Carlos García Garino[1,2] and Alejandro Zunino[3]

[1] Instituto para las Tecnologías de la Información y las Comunicaciones (ITIC), UNCuyo, Mendoza, Argentina {ssalinas,cgarcia}@itu.uncu.edu.ar
[2] ITIC & Facultad de Ingeniería, UNCuyo, Mendoza, Argentina
[3] ISISTAN, Facultad de Ciencias Exactas, UNICEN, Tandíl, Argentina
azunino@exa.unicen.edu.ar

Abstract. The application area of peer-to-peer (P2P) networks has grown in the last years. In a recent work the authors introduced a peer-to-peer overlay network based on the super peer model. In order to implement this overlay network the development of core P2P functions is required. These functions include peer organization, identification, message routing and fault tolerance among others. Different P2P frameworks were considered including JXTA and JGroups. In general, these frameworks provide a set of basic functionalities to facilitate P2P application development. However, they require extensive internal modifications for developing our P2P, because they already define the basic building blocks without the required flexibility. This paper introduces a configurable overlay network architecture for the development of core P2P functions. The architecture is based on the processing of messages, events and states. Thus, it is possible to translate a protocol definition based on a Finite State Machine into this architecture. On the other hand, a dynamic behavior definition is supported in order to introduce flexibility. Both features are suitable to implement a P2P overlay network.

1 Introduction

Peer-to-peer (P2P) networks have become a wide research area in the last years. Different P2P overlay networks have been proposed to satisfy different requirements. In a previous work [1] the authors presented Q-Fractal, a proposal for a P2P overlay network topology. The topology proposed follows the Gnutella 2 [2] super peer model. QFractal organizes peers in Federations leaded by a super peer. The network has different levels of interconnection in order to scale the system. Peer identification depends on its position in the overlay network. Thus, a peer can estimate the overlay network structure and calculate routes between peers based on a simple calculation.

In order to implement and validate Q-Fractal different frameworks were considered.

JXTA [3] is a framework composed of six XML-based protocols that standarize the manner in which peers self-organize into peergroups, publish and discover peer resources, communicate and monitor each other. The JXTA protocols enable developers to build and deploy interoperable services and applications.

JGroups [4] is a toolkit for reliable group communication. JGroups is composed of three components: a channel used by application programmers to build group communication applications, building blocks that are layered on the top of the channel and finally a protocol stack. The protocol stack contains a number of protocol layers in a

bidirectional list. All messages sent and received over the channel have to pass through all protocols. The composition of the protocol stack is determined by the creator of the channel.

JXTA and JGroups are not suitable for the Q-Fractal implementation because they provide support for P2P application development instead of P2P protocol development. These frameworks have already defined the way groups are created and how messages are routed.

On the other hand, PROST [5] is a programmable infraestructure based on a key-based routing layer of a structured P2P network. This layer provides the main functionality of efficiently mapping object identifiers to live nodes and locating them in the P2P network. This infraestructure allows dynamic loading of code modules named Peerlets onto nodes of the P2P overlay. These modules implement the application-specific functionality using the key based routing layer.

In [6] a framework for P2P application development is described. This work implements an abstraction framework that attempts to encourage developers to build P2P applications. It provides layers of abstaction that further isolate the developer from the complexities of the underlying P2P technology. The framework is not suitable for applications that require controlling the network or customizing the communication mechanisms between peers.

Both frameworks aim to support P2P application development instead of core P2P components development such as routing mechanisms. In order to implement the Q-Fractal model a software architecture was designed.

This paper introduces a configurable overlay network architecture that allows creating configurable peer-to-peer nodes. The main goal is to provide a flexible framework for the QFractal implementation. Its operation is event-driven, based on the processing of messages, events, states and the definition of plans of actions in response. Our architecture relies on finite state machines for allowing flexible configuration of peers. In addition, our architecture allows defining and configuring different behaviors for peer-to-peer nodes. These behaviors are specified by using rules related to topology maintenance, message routing, peer affiliation and identification, fault tolerance criteria and so on. The set of rules that define a behavior is called a profile.

The user applications may create one or more instances of a peer-to-peer node in a single host. The use of different profiles makes a peer-to-peer node instance respond in different ways to the same events or state changes. Thus, it is possible to run more than one node instance working under different profiles.

The rest of this paper is organized as follows. In section 2 the architecture is presented. The main component of the architecture, the core module abstraction is introduced in section 3. In section 4 examples based on the architecture are presented and finally section 5 introduces the conclusions and future works.

2 Architecture

Peers interaction in an overlay network is based on rules that define message routing, peer identification, fault tolerance criteria, peer affiliation, etc. These rules define the actions to be taken in response to events created by peers interactions.

The set of rules, actions, events and its relations in our model is named profile. Thus, peers working under a profile named P may use a binary tree overlay while peers working under a profile named Q may use a super peer overlay.

QFractal implementation requires a profile definition which relies on the architecture proposed in this paper. The architecture is based on the following components:

- Messages: a data structure for communicating and exchanging information between peers.
- Instruction: a set of actions and events that implements services to user applications.
- Event: a data structure that represent events created by peer interactions.
- Action: an implementation of functionalities such as send an UDP message, create a TCP connection, etc.
- Plan: a set of actions required to obtain an expected result.
- Result: a set of data represented under any standard to provide results to user applications.
- Request: a data structure that includes a set of parameters and allows modules to interact with the behavioral unit.
- Inner state: set of variable that describe a peer state at a moment of time.

All these components allow peers to interact with other peers and user applications. The interaction between peers is based on message exchange. On the other hand, peers provide functionalities to user applications. Both activities consist on processing instructions and messages, which may change the inner state and trigger new messages or events to be processed.

All these components flow through the architecture shown in figure 1. The terms actions and plan are interchangeable. The architecture modular design allows assigning common task and responsabilities to every module to introduce flexibility. It also provides all elements that support Finite State Machine expressions, a frequent tool used in protocol design. There are three groups of modules: the user application interface, the

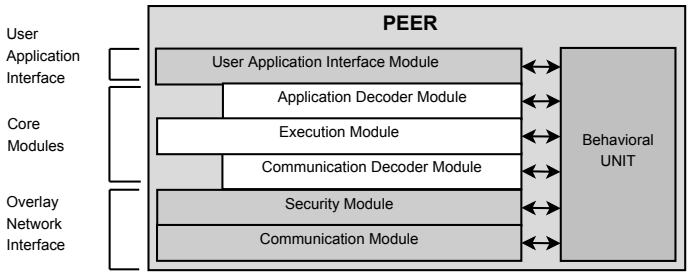

Fig. 1. Configurable Overlay Network Architecture

overlay network interface and the core modules. In the next sections the architecture modules are presented.

2.1 User Application Interface

Modules in this group are responsible for the integration of the overlay network services with user applications. This proposal include only one module in this group but in future works it is possible to add new modules in order to respond to new requirements.

The user application interface module provides to users an abstraction of peer-to-peer implementations. A set of instructions can be defined using the combination of events, actions and messages provided by the behavioral unit.

2.2 Overlay Network Interface

This group implements the funcionalities required to communicate peers in the overlay network. It allows the interchange of messages between peers and provides security mechanism for peer communication.

2.2.1 Security Module: this module provides the mechanisms required for a secure interchange of messages and information. As mentioned before, messages are a data structure thus it is posible to create messages from outside the peer and send them into the overlay network. It is necessary to avoid the manipulation of peer behaviour by messages not generated by peers. The security module implements message encryption and decryption to provide a secure communication. On the other hand, this module provides message filtering to allow or deny communication with nodes in the overlay network.

2.2.2 Communication Module: this module is responsible for the management of reliable connections with other peers. Besides, it provides mechanisms for information interchange such as file transfer. The messages sent can use a reliable point-to-point connection with other peers or multicast messages to a set of peers. All incoming message or information are sent to the security module in order to be processed by this module.

2.3 Core Modules

This group of modules implement the rules that define a profile. These rules determine the actions to be taken in response to events or inner state changes. In order to obtain this set of actions, a request is sent to the behavioral unit, which responds with a plan to be executed. These modules have an inner state that may change as consequence of the creation of events or the execution of actions. All core modules share the same components described in section 3.

2.3.1 Application Decoder module: the application decoder module have the responsability of creating events and actions to be processed by the execution module. It translates instructions into events and actions required by the execution module to obtain an expected result.

2.3.2 Communication Decoder module: this module has as input messages that are processed to create events, actions or trigger inner state changes. When a new message is received, a request for actions is sent to the behavioral unit. These actions may include altering the module state, creating new events, providing a plan to be processed by the execution module.

2.3.3 Execution Module: this module is responsible for the execution of plans provided by the application or communication decode modules. The processing of plans may create events, messages or results. This module can access to the overlay network interface to send messages or results over the overlay network. On the other hand, it is possible to sent results to the user application interface.

2.4 Behavioral Unit

The behavioral unit is one of the main components of the architecture. When one of the core modules receives an event or detects a state change, a request to the behavioral unit is sent. This unit is responsible for the decision of what to do in response to those events or environmental changes. The orchestration of all architecture components is accomplished by this unit.

2.5 An overall view of the architecture

Figure 2 shows the interaction between modules. To simplify the description, the interactions between modules with the behavioral unit have been represented in the left side of the figure. The core modules represented with white boxes have similar outputs. When any core module processes an input it may create new events or inner state changes. This triggers a request to the behavioral unit in order to get a set of actions to be executed. When an instruction is decoded by the application decoder, a set of actions

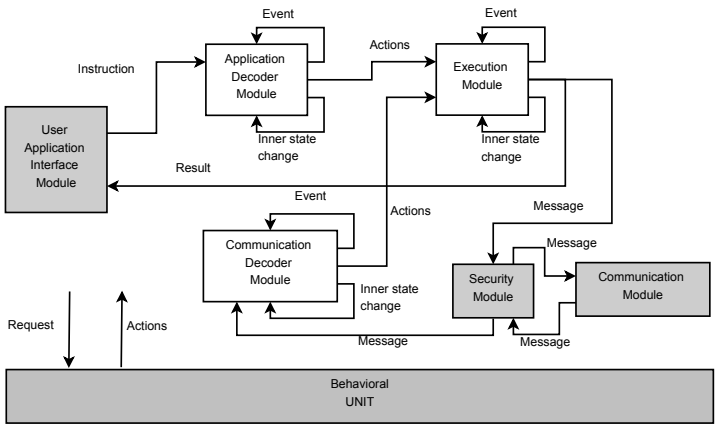

Fig. 2. Overall View of the architecture

is sent as input to the execution module. The execution module may create a message to be sent by the communication module after being processed by the security module. Also, this module may create a result which is sent to the application user interface.

On the other hand, incoming messages after being processed by the security module are sent to the communication decoder module. This module will create a set of actions to be processed by the execution module. The core modules request the behavioral unit to get a set of actions to be executed. These set of actions can be altered to change the response to an event or inner state change.

A more detailed explanation of how the core modules work and its inner components is presented in the next section.

3 Core Module Abstraction

The core module abstraction is a set of components that work based on the presence of events, inner state changes or plans to executed. The components are the following:

1. Worker: is a thread created for the processing of actions or events generated by the interactions with other modules. The actions to be taken by a worker are requested to the behavioral unit.
2. Controller: is a thread responsible for the processing of module state changes. When state changes the controller requests to the behavioral unit the set of actions to be executed.
3. Module state: is a set of variables and values which identify the core module state. The execution of actions may change the module state and trigger or not actions or new events in response to those changes.

The core module is responsible for wokers management and is able to create, pause and delete a worker. When an event, action or message arrives the module looks for the worker responsible for processing that event. If the worker does not exists the core module creates a new one. Thus, for every node or user application interaction there is a worker for processing that interaction.

3.1 Workers

As mentioned before workers are responsible for the interactions with other modules. The main worker components is shown in figure 3 Workers components are the following:

1. Event queue: a queue of events coming from the core module.
2. Event decoder: it reads the next event to be processed if the queue is not empty. When an event is read from the queue a request to the behavioral unit is sent in order to get a plan to be executed in response to this event.
3. Plan queue: a queue of plans provided by the event decoder. This queue holds the plans to be processed by the plan processor.
4. Plan processor: executes every action of a plan and alter the environment if is necessary.

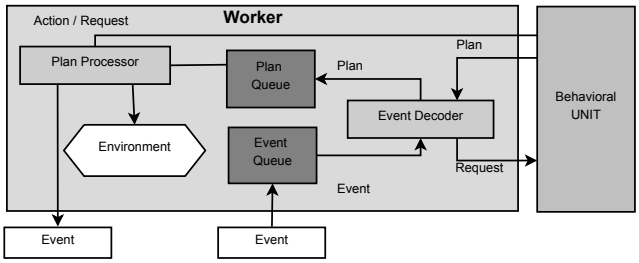

Fig. 3. Worker

5. Environment: a set of variables and values that are read or written by the plan processor when a plan is executed.

A worker receives one or more events that are enqueued. The event decoder takes events from the queue and requests the behavioral unit for a plan to be executed. Once the event decoder has obtained a plan, it is sent to the plans queue.

The plan processor executes plans from the queue altering the environment and creating other events if necessary. On the other hand plans from another module are stored in the plan queue in order to be executed.

3.2 Controllers

A controller is a thread that analyzes the module state and executes a plan in response to state changes. Figure 4 shows the main controller components.

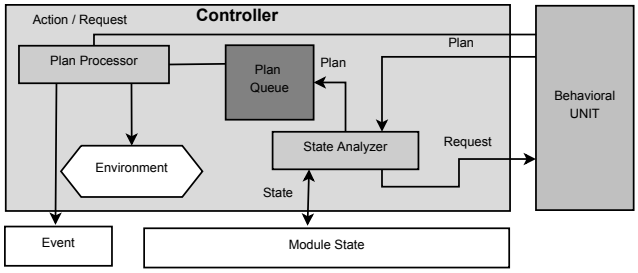

Fig. 4. Controllers

The controller components are the following:

1. State processor: this component controls state changes and requests plans to the behavioral unit.
2. Plan queue: a queue where plans are stored to be processed.
3. Plan processor: executes every action of a plan and alters the environment if necessary.

4. Environment: a set of variables and values which are read or written by the plan processor when a plan is executed.

A controller is responsible for executing a set of actions when the module state changes. When the controller is aware of any state change, it requests a plan to the behavioral unit and stores the plan in the queue. Every plan in the queue is processed by the plan processor and may alter the environment or create new events.

There is a difference between the environment and the module state. The environment scope involves a worker, thus an action updates or reads values used by other actions. On the other hand, the module state keeps the global state of the module.

3.3 Plan Processor

The plan processor is a set of components that receives as input a plan. While the plan is processed it may generate events or modify the worker or controller environment. Figure 5 shows the plan processor architecture. The plan processor components are:

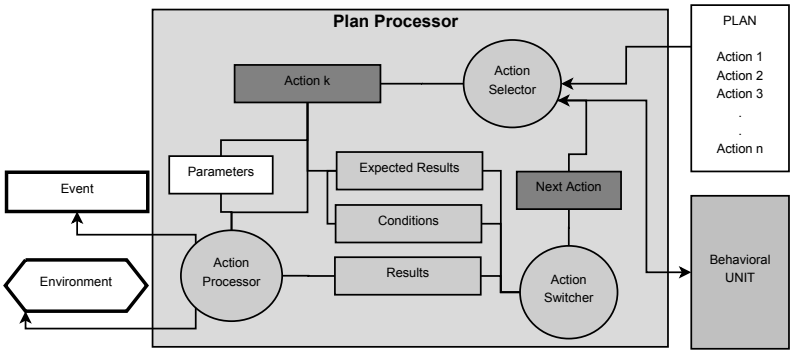

Fig. 5. Plan Processor

1. Action selector: this component selects the next action to be processed.
2. Action processor: executes a software implementation for an action.
3. Action switcher: evaluates a set of parameters to determine the next action to be executed.

In order to understand how a plan processor works a more detailed explanation of plan and actions is introduced in the next subsection.

3.3.1 Plan: a plan is a set of one or more actions to be executed by a plan processor. As mentioned before, the execution of a plan may create new events, modify an environment or change the state of a core module. For instance:

$$\textbf{Plan} = [\, action_1, action_2, action_3, \ldots, action_n]$$

Every action has information for evaluating the results generated once the action have been executed. Then, the action switcher may suggest the execution of another action or continue with the next action in the plan. Thus, a plan is integrated by a set of static actions and a set of dynamic actions. Static actions are the enumerated in the plan definitions whilst dynamic actions are loaded after the evaluation of an action result.

3.3.2 Action: an action implements a software functionality executed by the action processor. The implementation of an action is requested to the behavioural unit. For instance, let us consider an action sendJoinRequest, which may have one or more implementations. The behavioral unit will return the implementation that have been selected in the profile definition. An action is conformed by the following components:

1. Parameters: set of values required by the action implementation.
2. Expected results: set of values expected to be reached once the action is executed.
3. Results: set of values generated by the execution of an action.
4. Performance result: value resulting of the evaluation of the results obtained after an action execution and the expected results for that action.
5. Conditions: a logical expression which defines what the next action to be taken having as input the result of a performance evaluation is.

3.3.3 Action processor: the action processor execute a software module represented as an action in order to achieved a desired effect. For instance, establishing a communication channel with another peer. The action execution may generate events or alter the values of the environment variables. Actions are provided by the action switcher introduced below.

3.3.4 Action switcher: an action switcher is responsible for the selection of an action as result of the evaluation of switcher input parameters. The evaluation process is executed by the switcher and the result of this process is an action or an end of process signal. The input parameters for an action switcher are:

- Results obtained
- Expected results
- Conditions

The evaluation process is the following: let us name p the performance result, xr the expected result for an action, r the result obtained after the action execution, $Compare(xr, r)$ a function that is 1 if $xr = r$ otherwise 0, then:

$$p = \frac{\sum_{i=0}^{n} Compare(xr_i, r_i)}{n} \tag{1}$$

Based on the perfomance evaluation described by the equation 1, the switch selector compares the result with the set of conditions, represented in 2, in order to determine

the output for this process.

$$Conditions = \begin{cases} v_1 \leq p \leq v_2 & action_1 \\ v_3 \leq p \leq v_4 & action_2 \\ \vdots \quad \vdots & \\ v_{n-1} \leq p \leq v_n & action_n \end{cases} \quad (2)$$

3.3.5 Action Selector: the action selector receives as input the result of the action switcher. If this result is eop then the action selector picks the next action in the plan, if it exists, otherwise the plan execution finishes. If the result contains an action, then the action selector sends that action to the action processor and provides the arguments necessary to the action switcher to determine what to do next.

4 Examples based on the architecture presented

The architecture proposed has been partially developed using the Java lenguage. Thus, the peers deployment will be platform-independent. Only the basic functionalities have been programmed in order to test the examples introduced in subsections 4.1 and 4.2.

At this preliminar stage it is necessary an easy deployment of peers in order to test peer configuration and communication. For this reason a network virtualization was adopted as a testbed.

A network of twenty peers was created under two host. Each host virtualizes a subnet of ten peer each connected by a virtual bridge. Two features were intended to be tested, the support for a FSM translation into the architecture and the redifinition of plans for a profile. Both examples are presented in the next subsections.

4.1 An example of a profile definition based on FSM representation

It is possible to define the rules that govern peer behavior based on states and events. Figure 6 shows a finite state machine (FSM) that models a peer behavior when joining to a peer-to-peer overlay network such as Q-Fractal.

In the first place, when the peer start up sends an idRequestMessage broadcast and waits for a response. If another peer is present then this peer will assign a peer id to the new peer. This event is represented in the FMS with the state peer configure. If no peers are present in the network the new peer will try a number of times to get an id. After that it will request an id from the behavioral unit and will create a number of ids to supply when new peers attempt to join the network. The id value and the numbers of ids to supply are configuration parameters defined by the user.

If a peer request arrives then will assign a peer id, answer the request and wait for an acknowledgement message. If no acknowledgement arrives the id will be available for other requests.

When no peer ids are left to be assigned, the event hasNoMoreIds is threw. The sequence of actions of these last states have not been included in order to simplify the example.

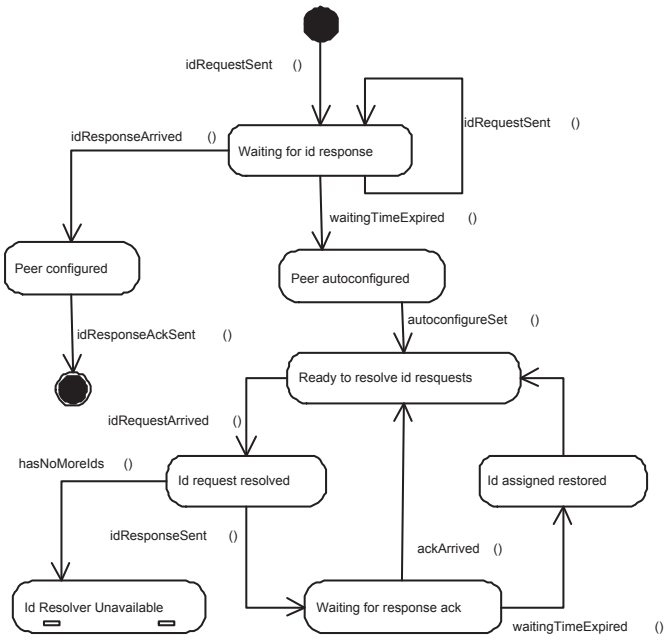

Fig. 6. Finite State Machine

The FMS is then translated to a set of values that defines what to do when certain events are present. For instance, the configuration of the event idRequestArrived may be the following:

- Profile: QFractal
- State: Ready to resolve id requests
- Plan: resolve id request
- Actions: getNextIdAvailable
- ExpectedResult: id>0
- Condition1: if evaluation of results = 1 then nextAction is sendIdResponse.
- Condition2: if evaluation of results = 0 then nextAction is createEvent hasNoMoreIds.

As mentioned before, when an id request message arrives the communication decoder module will create the event idRequestArrived. Then, it will request the behavioral unit for a plan that will be send to the execution module. If the execution module state is ReadyToResolveIdRequest then it will change its state to AnalyzingIdAvailability. Then, the plan will be executed and a message response with the peer id assigned will be sent. If the execution module state is AnalyzingIdAvailability the plan will be enqueue to be process.

Finally, if the execution module state is IdResolverUnavailable, the execution module will request the behavioral unit for a plan.

In the next section an example of how to alter the behavioral unit response will be introduced.

4.2 An example of plan redefinition

The following example shows how to redefine the peer behavior in response to the same event. The behavioral unit, provides a plan to be executed in response to certain events. A plan has been defined as a sequence of actions that is possible to alter. For instance, lets consider the following situation:

- The event named *peerIdRequest* is thrown.
- A worker requests a plan to the behavioral unit.
- The actions, part of the plan are: check peer id availability and send a response.

Now, suppose an user application, for security reason, denies the access to a set of IP address.

A possible redefinition of the plan is the following: the new version of the plan is, control IP and if it is ok check peer id availability and send response. Thus, a peer changes its response to the event *peerIdRequest*.

On the other hand, a full plan can be replaced by another version. For example, a criteria for fault tolerance may be defined by a plan. The result of the execution of this plan can be evaluated by any user application. According to the evaluation result the current version of that plan can be replaced by a new version. This feature facilitates the peer's performance improvement and increases the system flexibility.

5 Conclusions and future works

This paper has introduced a configurable overlay network architecture in order to provide a framework for the implementation of QFractal. The architecture is based on the processing of events and state changes. Thus, it is possible the translation of a protocol specification using FSM into the architecture introduced in this work.

On the other hand, the architecture supports configuration changes at run time. This feature introduces flexibility and adaptability of peers when defining their behavior.

The framework resulting of the architecture's programming facilitates the QFractal implementation according to the characteristics introduced in this paper.

In future works, the QFractal specification will be defined and be translated into the architecture presented.

References

1. Sergio Ariel Salinas, C.G.G., Zunino, A.: Q-fractal: A proposal for a p2p overlay network topology. 10th Argentine Symposium on Computing Technology AST 2009 (2009) 113–126
2. Shicong Meng, C.S., et.at. In: Gnutella 0.6. Volume 3841 of LNCS. Springer Berlin / Heidelberg (2005) 189–200
3. Gradecki, J.D., Gradecki, J.: Mastering JXTA: Building Java Peer-to-Peer Applications. John Wiley and Sons (2003)
4. Bela Ban, V.B., et. at.: Jgroups (2002-2010) `http://www.jgroups.org/`.
5. Portmann, M., Ardon, S., Senac, P., Seneviratne, A.: Prost: A programmable structured peer-to-peer overlay network. Peer-to-Peer Computing, IEEE International Conference on **0** (2004) 280–281
6. Walkerdine, J., Hughes, D., Rayson, P., Simms, J., Gilleade, K., Mariani, J., Sommerville, I.: A framework for p2p application development. Comput. Commun. **31**(2) (2008) 387–401