Owlet: Event-based Interaction Language for Constructing Large-Scale Distributed Systems

Abstract

There is an increasing need to build large-scale distributed systems over the Internet infrastructure. However, due to the autonomy and dynamic of resources in these systems, the interactions among distributed components and services have not been well supported in the existing programming languages. We present Owlet, an interaction language based on peer-to-peer content-based publish/subscribe scheme. Owlet abstracts the Internet as an environment for the roles to interact, and uses roles to build a relatively stable view of resources for the on-demand resource aggregation. It provides language constructs to 1) use distributed event driven rules to describe interaction protocols among different roles, 2) use conversations to correlate events and rules into a common context, and 3) use resource pooling to do fault tolerance and load balancing among networked nodes. We have implemented an Owlet compiler and its runtime environment, and built several Owlet applications, including a peer-to-peer file sharing application. Experimental results show that the separation of resource aggregation logic and business logic significantly eases the process of building large-scale distributed applications.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – concurrent programming structures; D.3.2 [**Programming Languages**]: Language Classifications – concurrent, distributed, and parallel languages; D.1.3 [**Programming Techniques**]: Concurrent Programming – distributed programming.

General Terms Languages.

Keywords resource aggregation, interaction protocol, event-driven rules, conversation correlation, resource pooling

1. Introduction

The Internet can provide unprecedented amount of resources by containing a vast number of entities distributed all over the world. Research has shown that these resources, e.g. content, storage, CPU cycles, bandwidth, and human presence, are vastly underutilized [12]. On the other hand, software systems need more and more resources to accomplish their tasks, e.g. the recently launched Large Hadron Collider (LHC) will produce roughly 15 petabytes of data a year that need to be analyzed by scientists around the world. There is an increasing need to build large-scale distributed system over the Internet infrastructure to utilize various idle resources to get better overall performance, or simply get the task done. For example, the LHC@Home project uses its participants' computer idle time to simulate how particles will travel in the tunnel; many peer-to-peer file sharing and video streaming applications use idle bandwidth and storage of its peers to gain better individual experience.

However, the interest and behavior of the entities in the Internet may greatly vary throughout the lifetime of a software system, e.g. resources may spontaneously turn busy, or even leave the network. One of the key problems emerged is how to on-demand aggregate resources in the Internet environment, i.e. how to collect, organize, and comprehensively utilize resources without global information, and form a relatively stable view of individually transient resources for applications [18]. Resource aggregation is achieved through interactions between distributed components and services. One component needs to collect available services according to certain criteria by sending messages to other components, and use the response information to organize the services for load balancing, and when one node fails during utilization, it may contact another node to do the failed task.

Due to the autonomy and dynamic of component and services in large-scale distributed systems, the interactions among distributed components have not been well supported by existing languages and platforms. Traditional distributed component technologies, e.g. Enterprise JavaBeans (EJB) and CORBA Component Model (CCM), typically use naming services to register components during system deployment, and the interactions among components are statically coded. They are suitable for intranet environment, where resources are usually dedicated to the components. However, they do not scale well to the Internet, where node failure will become a common case.

Web services support loosely coupled distributed systems in service discovery and service execution, and interactions among services can be specified using service composition languages, e.g. WS-BPEL [2]. However, load balance in service selection and fault tolerance in service execution need to be explicitly programmed, e.g. though XL [11] can automatically retry a failed action, it can not automatically schedule the action to another available service.

Software agents are autonomous, proactive, and they interact with each other to achieve their goals. Agents can model the autonomous and dynamic resources on the Internet, thus they are well suited for building large-scale distributed systems [14]. The existing agent platforms and interaction languages such as OWL-P [8] and IOM/T [9] focus on describing interactions among different roles of agents, while manual configuration or extra coding are needed to discover agents, e.g. using directory services.

In this paper, we present Owlet, an interaction language based on peer-to-peer content-based publish/subscribe scheme. Owlet uses agents to represent resources on the Internet, and characterizes the agents into different roles of interactions for resource aggregation in large-scale distributed systems. The agents are organized in a peer-to-peer manner, which are capable of adapting

SIGPLAN'05 June 12–15, 2005, Location, State, Country.

to failures and accommodating transient populations of resources, while maintaining acceptable connectivity and performance [3]. Unlike the existing programming languages for building distributed systems, which typically use point-to-point message passing as their communication primitives, Owlet uses content-based event publish/subscribe interaction scheme, which is claimed to provide the loosely coupled form of interaction required in large-scale settings. An event is asynchronously propagated to all subscribers that registered interest in that given event. This event-based interaction style brings full decoupling in time, space, and synchronization between publishers and subscribers [10].

Owlet separates the concerns of constructing distributed systems into the resource aggregation logic and the business logic. The aggregation logic is specified by the interactions among roles to collect and organize available resources. Each role contains rules that guard on certain events to take some actions. The actions may be the publishing of another event, or invoking the business logic. By using roles to characterize the underlying resources, programs can aggregate resources by interacting with the corresponding role without the knowledge of individual agents that represent the resources. Although an individual agent may be transient due to the autonomic and dynamic nature of the resource it represents, their role as a whole forms a relatively stable view of the resources. The Owlet runtime is responsible for selecting individual resources in a proper way, e.g. with load balancing and fault tolerance. Once the resources are selected, Owlet delegates the business logic to the existing programming languages and component technologies, which are well suited for processing resources in a determined context.

Our main contributions are language constructs that 1) use distributed event driven rules to describe interaction protocols among different roles to facilitate collecting, organizing, as well as utilizing of resources on the Internet, 2) use conversations to correlate events and rules into a common context to better specify the interactions in a hierarchical way, and 3) use resource pooling to do fault tolerance and load balancing among networked nodes based on the event publish/subscribe interaction scheme. With these constructs, large-scale distributed systems can be elegantly built with a clear separation of resource aggregation logic and business logic.

We have implemented an Owlet compiler and its runtime environment, and built several Owlet applications, including a peer-to-peer file sharing application. Experimental results show that the separation of resource aggregation logic and business logic significantly eases the construction of large-scale distributed systems. For example, a program for the peer-to-peer file sharing application is about 140 lines in Owlet, and the underlying business logic component is very straightforward to implement. Experiments also show that Owlet programs can achieve a high level of fault tolerance and a certain degree of load balancing among networked nodes even when individual resources are fairly unstable.

2. Owlet

Owlet is an interaction language, which specifies interactions among distributed entities via network communications. Interactions are mandatory for the entities to utilize the resources of others in distributed systems. Most of the traditional approaches for building distributed systems use point-to-point message passing as the communication primitive to specify the interactions, which leads to rigid interaction schemes. As in message passing, the sender of a message must always know the receiver, and when a message needs to be addressed to multiple receivers, it is usually sent one by one to each receiver. These rigid interaction schemes force programmers to write extra code or do manual configuration to locate the receivers, thus they can not scale well to build large-scale distributed systems, especially those over the Internet.

We focus on describing the interaction among entities by interaction protocols, and leave the local operations on resources of each entity to existing component programming technologies. Protocols involve several roles and address specific purposes of applications. They emphasize the essence of interactions and omit local details. Instead of modeling interaction protocols in terms of point-to-point message passing, we use content-based event publish/subscribe to capture the interaction scenarios of the participants to accommodate the open, dynamic nature of the Internet.

We use agent as the basic notion to mirror the autonomous and dynamic entities providing or requesting resources in the Internet. Agents are autonomous, proactive, and situated in a particular environment. They interact with each other through sensors and actuators [14]. Agents behave autonomously by following their rules, which use the sensors to perceive the environment to get inputs, and use the actuators to affect their environment, which will probably become the inputs of other agents. Based on this notion, Owlet uses the publish/subscribe interaction scheme to decouple event subscribers from event publishers, which will turn the Internet into an environment to provide the surrounding conditions for agents to exist, the mediation of agents' interactions, and access to resources [20].

We use roles to characterize agents representing particular resources, e.g. `Provider` and `Requester` are two roles in a file sharing applications, and a `Requester` agent may interact with several `Provider` agents to download files. Agents in the same role have the same behavior rules to interact with others. Owlet specifies the interactions by using roles as the first class entity, because roles can give programmers a relatively stable view of the resources by representing the agents as a whole, while individual agents may be transient. The relationship between roles and agents are dynamically bound in Owlet, i.e. an agent can join and quit a role at runtime either under user request, or by its rules.

Figure 1 shows the concept model of an Owlet program, which defines a set of resource metadata schema and involves several roles to describe the interactions. Owlet uses XML data model to represent states, variables and events. It supports primitive types such as Boolean, integer, floating point number and string. Composite types and list types can be structured hierarchically based on these primitive types. Owlet expressions use a subset of XPath expressions [6]. The resource metadata schema defines data types for describing the properties of the resources. They will be used by the roles to define states, variables and events in the interactions.

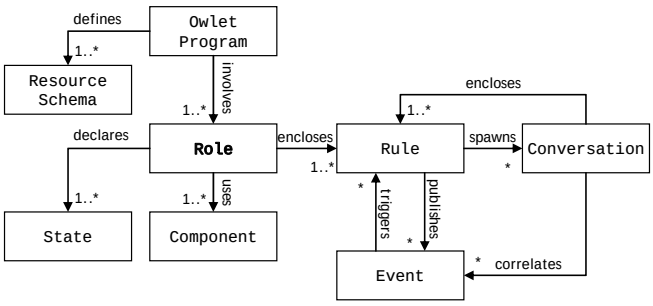

Figure 1. Owlet concept model.

Roles are composed of three parts: state variable declarations, interaction rules and component references. The rules describe patterns of interested events and corresponding actions to execute when an appropriate event is triggered. Rules can spawn conver-

sations that further correlate events and rules into a common context. The local operations on resources are delegated to the components.

Based on this concept model, the main language features of Owlet are illustrated in the following three subsections.

2.1 Event-driven interactions

Owlet uses distributed event driven rules to describe interaction protocols among different roles. All the rules are concurrently guarding on interested events from the environment to take corresponding actions. When an event is published to the environment, it will trigger all the rules whose patterns can be matched by the event. So an interaction is composed of two parts: the publishing of an event, and the rules that guard on the particular event.

Publishing events are basic statements in Owlet. An event statement is composed of an event name and an arbitrary number of parameters, e.g.

```
provide(avails).
```

The above example means publishing an event named "provide" with one parameter whose value comes from variable `avails`. Note that the event publisher does not need to know the information of the event receivers, and a single event may usually notify multiple receivers. However, the event does have information about its publisher, together with its name and parameters.

The declaration of a rule starts with the keyword `when`, and is also composed of two parts: a pattern of the events to be matched and statements (e.g. publishing an event) to execute when an event matches the pattern. A pattern can specify various constraints on events it matches, e.g. event publisher, event name, and the type or value of its parameters. Figure 2 shows an example rule. Its pattern will match events that 1) is published by an agent of role `Provider`, 2) is named "provide", and 3) has one parameter whose type is an integer list.

```
when ( Provider p: provide(int* avails) ) {
        ...
}
```

Figure 2. An example rule.

After the matching, the unbound variables in the pattern will be bound to corresponding values from the event, which then can be used by the statements of the rule. For instance, in Figure 2, `p` will bind to the identity of the event publisher, and `avails` will bind to the first parameter of the event, which must be an integer list.

```
when ( self: take(File file) ) {
        ...
}
```

Figure 3. A rule that matches a particular event publisher.

An agent has a global unique identity to distinguish from each other, e.g. a UUID. Patterns can specify the constraints of the event publisher by its role or by its identity. The keyword `self` is used to represent the identity of the current agent. For example, Figure 3 shows a rule whose pattern will match events named "take" with one parameter of type `File`, and are published by the agent itself.

```
when ( Requester r: request(self, Piece need) ) {
   ...
}
```

Figure 4. A rule that simulates point-to-point messaging.

This publish/subscribe interaction scheme decouples event publishers from event subscribers. Moreover, point-to-point messaging can still be achieved by explicitly specifying the identity of the receiver in the parameters, and use a rule with pattern that matches the parameter to the identity of the agent, as shown in Figure 4. When an agent of `Requester` role publishes an event, e.g.

```
request(p, need).
```

The rule in Figure 4 will only be triggered for agent whose identity equals to p.

This event publish/subscribe interaction scheme is more expressive than point-to-point message passing. Method calls in object-oriented paradigm always imply a reference to the invoked object, e.g. keyword `this` in Java. In addition to the keyword `self` representing the identity of the current agent, the event-driven rules in Owlet always bear the identity of the publisher of an event, which is crucial for providing a loose coupled way to describe interactions in large-scale distributed systems.

2.2 Conversations

Owlet uses conversations to correlate events and rules into a common context. We can describe interactions by specifying all the rules flatly. However, this flat structure is not convenient for specifying complex interactions of large-scale distributed systems, where certain steps of the interactions may depend on other steps, i.e. the interaction rules may have dependencies. Conversations are used to represent these dependencies by arranging rules in a hierarchical way.

Figure 5 demonstrates part of the rules of the `Requester` role in a file sharing application. The rules are arranged into two levels by a conversation. These bring two benefits: 1) by building these hierarchies, the dependencies among rules are enforced to reduce undesired interactions; 2) the variables in the outer level (e.g. `file`) become context variables and can be easily accessed from the inner levels.

```
// role Requester
when ( self: take(File file) ) {

  converse {

    when ( self: request(file.digest) ) {
         ...
      request(file.digest, needs);
    }

    when ( Provider p: provide(file.digest, int* avails) ) {
         ...
    }
  }
}
```

Figure 5. Arranging rules hierarchically by a conversation.

Further, rules are used to specify the protocol of interactions, and there may be multiple instances of the same interaction protocol around different individual resources. For example, a `Requester` agent may request multiple different files from Provider agents simultaneously. The events and rules must be properly correlated to distinguish their interactions around different resources. In Figure 5, all the rules and events are augmented with a `file.digest` parameter, which is the digital digest of the file content that distinguishes one file from another, e.g. MD5 digest. However, this tedious correlation mechanism requires the programmers to write the correlation token many times wherever necessary.

Although the choosing of the token is specific to applications, the correlation mechanism can be simplified by managing the token in a conversation. As shown in Figure 6, the conversation is created by using the keyword `converse` with the `file.digest` as the token. All the events published in a conversation are auto-

matically augmented with the token of its conversation, and all the rules in a conversation will match events with the same token in addition to its pattern.

```
// role Requester
when ( self: take(File file) ) {

  converse (file.digest) {

    when ( self: request() ) {
        ...
      request(needs);
    }

    when ( Provider p: provide(int* avails) ) {
        ...
    }
  }
}
```

Figure 6. Managing the correlation token in a conversation.

When interactions in a conversation are complete, the context variables and rules can be released by using the keyword `conclude` in a rule.

By using conversations, the dependencies among different steps in a conversation are represented in a hierarchical way, and correlations within concrete processes of the interactions can be elegantly specified in Owlet.

2.3 Resource pooling

Owlet uses resource pooling to do fault tolerance and load balancing among networked nodes. It is extremely necessary when we are employing resources in the Internet, which may fail or leave spontaneously. And certain degree of load balancing is also necessary for tasks to be effectively processed.

Since Owlet uses event publish/subscribe interaction scheme, the publishing of an event are addressed to all agents that subscribe this particular event. This interaction scheme makes it direct to have multiple redundancy resources to fault tolerantly do a task. Although this makes the fault tolerance very easy, employing all available resources to do a job is indeed a waste of resources, which will eventually make the system unusable.

Owlet provides language constructs that specifies the number of the redundant receivers that an event will trigger in a conversation. As shown in Figure 7, the `checkPrime(sn)` event is annotated with an integer limiting the number of its receivers. This event is a request to use the CPU cycles of others to test whether `sn` is a prime number. In this case, there will be at most four receivers of this event. Each of them will do the test and publish the result using a `finish(result)` event.

```
converse (sn) {
  checkPrime(sn) -> 4;

  when ( Checker c: finish(int result) ) {
      ...
    conclude;
  }

  when ( self: fail() ) {
      ...
  }
}
```

Figure 7. Set the limit of redundancy event receivers.

Owlet uses a resource pool to do resource provisioning and scheduling for programmers. When publishing an annotated event in a conversation, a corresponding resource pool is created. A certain number of potential receivers are collected within a provisioning threshold or deadline. These receivers are sorted by some criteria, e.g. their current load or their capabilities. Then the events are delivered to the limited number of receivers. When the first event from these chosen receivers is captured by a rule inside the conversation, the resource pool is released.

It is possible that none of the receivers survives to finish the task. A built-in `fail()` event will be published inside the conversation to delegate the processing to the programmer, as shown in Figure 7.

The resource pooling capabilities of Owlet makes the process of fault tolerance and load balancing transparent for the programmers in building large-scale distributed systems.

3. Owlet Implementation

We have implemented an Owlet compiler and its runtime environment[1] based on Java, and an Eclipse[2] plug-in is also developed to help programmers to write/compile/debug Owlet programs. The compiler is built using JavaCC[3] to compile Owlet source code into Java source code. Figure 8 lists the syntax of Owlet's main language constructs.

```
Program   ::= Community [Schema] (Role)+
Community ::= community Name ;

Schema    ::= schema { (TypeDef)* }
TypeDef   ::= ID { (Type ID ;)* }
Type      ::= (bool | int | float | string | ID) [*]

Role      ::= role ID { (State | Component | Rule)* }

State     ::= Type ID [= Expression ] ;
Component ::= local ID URN ;
Rule      ::= When (, When)* { (Statement)* }

When      ::= when ( Pattern )
Pattern   ::= Source : ID ( [Param (, Param)*] )
Source    ::= self | [Name] ID
Param     ::= Expression | Type ID

Statement ::= Join | Quit | Event | Converse | Conclude
            | Declare| Assign | If | Enum | Break | Sync
            | Invoke | { (Statement)* }
Join      ::= join Name ;
Quit      ::= quit Name ;
Event     ::= ID ( [Expression (, Expression)*] ) ;
Converse  ::= converse ( Expression ) { (Statement)* (Rule)* }
Conclude  ::= conclude ;
Declare   ::= Type ID [= Expression] ;
Assign    ::= Expression = Expression ;
If        ::= if ( Expression ) Statement ;
Enum      ::= enum ID : Expression do Statement ;
Break     ::= break ;
Sync      ::= synchronized ( Expression );
Invoke    ::= ID . ID ( [Expression (, Expression)*] ) ;

Expression::= CreateExpr | PathExpr | InvokeExpr
CreateExpr::= Type ( [Expression (, Expression)*] )
PathExpr  ::= ID | PathExpr . ID | PathEpxr [ Expression ]
InvokeExpr::= ( Type ) ID . ( [Expression (, Expression)*] )
```

Figure 8. BNF syntax of Owlet's main language constructs.

The Owlet runtime environment provides necessary mechanisms to support Owlet language constructs. As shown in Figure 9, it has four layers: the transport layer, the overlay layer, the services layer and the role container layer.

The transport layer provides basic connectivity to support end-to-end communications among networked entities in the Internet. It provides a uniform interface for communication via various

[1] http://owlet-code.sourceforge.net

[2] http://www.eclipse.org

[3] http://javacc.dev.java.net

transport protocols such as TCP, UDP and HTTP. The transport layer abstracts the Internet as a peer-to-peer network, which is used by the overlay layer to form certain network topologies.

The overlay layer provides consistent routing capabilities for the peers by using overlay network protocols such as Chord [21] and FISSIONE [17], which are used by the services layer to provide resource aggregation services.

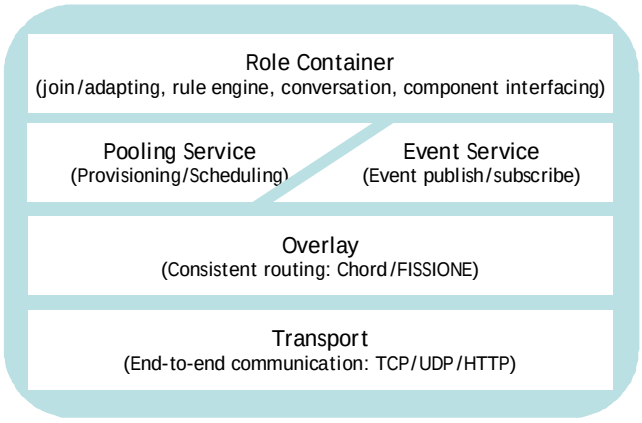

Figure 9. Owlet runtime implementation.

There are two services in the services layer: event service and resource pooling service. The event service provide efficient distributed content-based publish/subscribe support based on the underlying overlay network. Our implementation of the event service is based on the mechanism described in Ferry [25]. The resource pooling service provides resource provisioning and scheduling based on the event service and the transport layer to provide fault tolerance and load balancing support for Owlet programs.

The role container is the core of the Owlet runtime environment. First, it provides the dynamic binding of roles and agents, i.e. agents can join and quit a role at runtime. When an agent joins a role, it will automatically download the corresponding Java packages as described by the application's XML descriptor. Then it instantiates the states, bind the JavaBeans components, and assembles the rules to the rule engine. When an agent quits a role, all these components will be ~~dissembled~~ properly. Second, the role container has a rule engine that manages all the rules of the agents. When an event is matched to the pattern of a rule, the actions of the rule will be executed. Third, the role container manages the lifecycle of all the conversations and several built-in events. And finally, it manages the interfacing with the underlying components. We employ JavaBeans as the underlying component technology to handle the business logic. When invoking a method of a Java component, Owlet data types are automatically converted to corresponding Java types. And if there is a return value, it will be converted to the corresponding Owlet type. Since we use XML data model in Owlet, the conversion is straight forward to implement.

3.1 Event-driven interactions

Owlet programs describe interactions by using event publish/subscribe as the communication primitive. When an event is published, it should be propagated to all the subscribers that have subscriptions on such event. The subscription is based on the pattern that can match the event content, e.g. the role or identity of its publishers, its name, and the type or value of its parameters. A content-based publish/subscribe event service [10] is appropriate to efficiently support this kind of communications.

The event statement in Owlet will be compiled into Java code that publishes the event to the event service. The pattern in a rule statement is compiled to Java code that subscribes to the event service with a callback object that will be called when events that match the pattern are published to the event service. The callback object will schedule a thread from the thread pool to execute the actions of the rule.

Considering to build large-scale distributed systems over the Internet infrastructure, we have implemented a peer-to-peer content-based publish/subscribe event service according to Ferry [25], to accommodate the scalability of large-scale distributed systems. This implementation employs the overlay network to efficiently distribute the process of event matching across the network. As shown in Figure 10.

Peers in the overlay network cooperate to provide the content-based publish/subscribe event service. Events have properties such as name, publisher, parameters and etc. When subscribing to the event service, the subscription is delegated to a peer by randomly choose a property name of the event, and use it as the destination to route this subscription to the peer in the overlay network. By doing this, subscriptions are scattered among a determined set of peers. When an event is published, it will be routed to all the peers by using every property name of the event as the destination in the overlay network. Then, the matching of patterns and the event are processed at each peer holding the subscriptions, and they will notify the corresponding subscriber whose pattern matches the event. The consistent routing capability of the underlying overlay network ensures that all subscribers will eventually receive the right event.

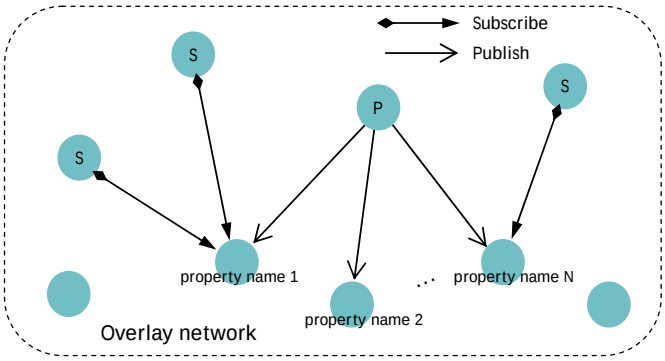

Figure 10. Content-based event publish/subscribe service based on peer-to-peer overlay network.

3.2 Conversations

Conversations are used to represent the dependencies between interaction rules by arranging them in a hierarchical way. They provide the context for the rules to access the shared variables and manage the correlation tokens of events and rules. A conversation can have sub-conversations, whose tokens are prefixed with the token of its parent conversation. All the conversations are managed by a conversation tree in the Owlet runtime environment. The first level rules of a role constitute the root conversation, which has a predefined root correlation token.

Rules are aware of their conversation context. They can access variables defined in the conversation and all its parent conversations. In the actions of a rule, new conversations can be created as sub-conversations of the current conversation, and all event statements will automatically augment the correlation token of the current conversation to the events.

When a conversation is created, it will subscribe all its enclosing rules to the event service, and augment those subscriptions

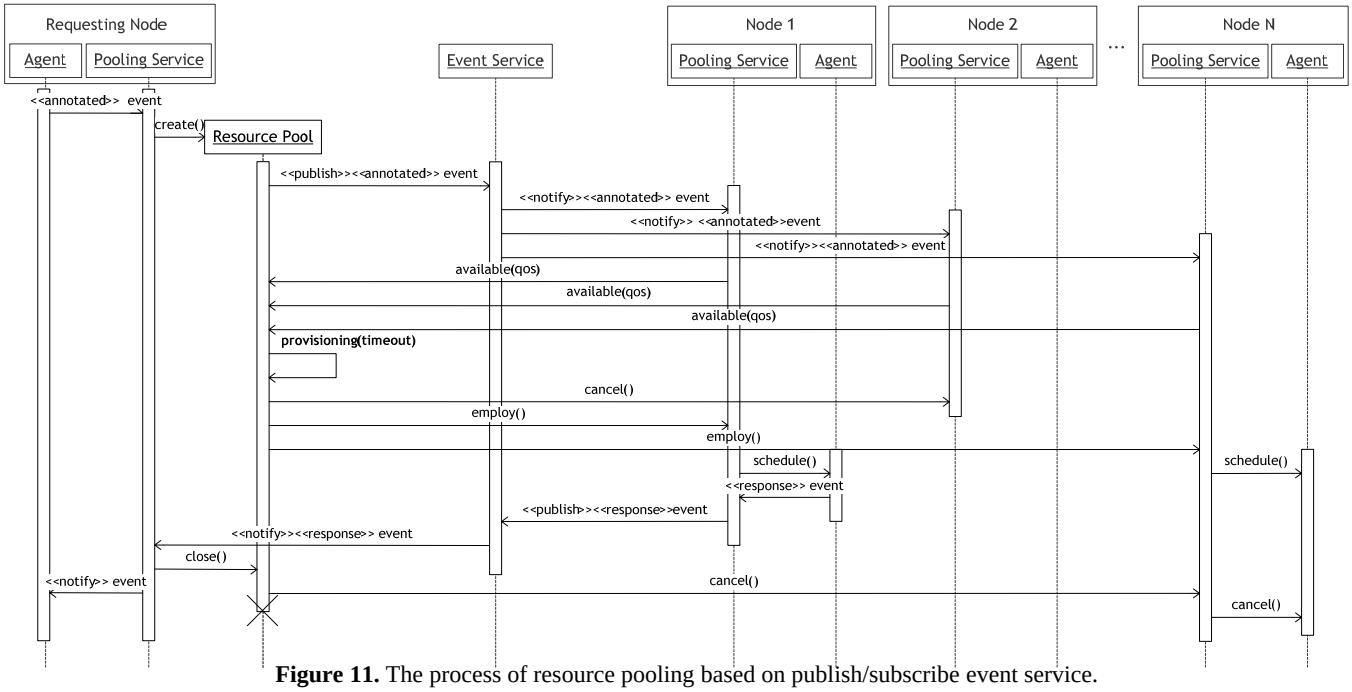

Figure 11. The process of resource pooling based on publish/subscribe event service.

with an extra constraint to match the event token with the conversation token. The matching of the event token and the conversation token is a prefix match, i.e. the conversation token must be a prefix or equal to the event token. This allows events in a conversation to trigger rules in its parent conversations.

When a conversation is concluded, it will first conclude all its sub-conversations, and then unsubscribe all its enclosing rules from the event service. Any running threads of the actions of its rules will be interrupted. Finally, all its context variables are released.

3.3 Resource pooling

The resource pooling mechanism in the Owlet runtime environment employs multiple redundancy resources for a single task to provide fault tolerance in large-scale distributed systems, where resources are usually tremendous for applications to utilize. It also provides a certain degree of load balancing among the resource according to their quality of service properties, e.g. system load, task queue size, and etc.

There are two phases in a resource pooling process: resource provisioning and resource scheduling. Figure 11 shows a resource pooling process based on the event publish/subscribe service. Some stereotypes are used in the sequence diagram to denote different operations on events.

The provisioning phase begins when executing an event statement with resource pooling annotations, which specify the maximum number of redundant receivers of the event. Instead of publishing it to the event service, the annotated event will be delivered to the pooling service, which will create a resource pool for holding potential receivers of the event. The resource pool is created with a globally unique serial number to distinguish from each other. Then, the event will be published to the event service with annotations about the serial number and transport address of the resource pool. The event service acts as normal, and notifies corresponding subscribers of the event. When a node receives the annotated event, it will not immediately schedule a thread to execute the actions of the corresponding rule. Instead, its pool service will send the requesting node an "available" message about its QoS properties. The resource pool of the request node is waiting for the messages within some threshold or deadline. And when the provisioning is finished, the resource pool of the request node will have collected a set of potential receivers of the event, together with their QoS properties.

Then the scheduling phase begins. The resource pool will sort all the potential receivers by evaluating their QoS properties. Next, "employ" messages will be sent to a number of the best receivers according to the initial event statement, and "cancel" messages will be sent to the rest. As shown in Figure 11, Node 2 and Node N are employed, while Node 1 is cancelled. The collected nodes will schedule the actions of the corresponding rule to execute when employed, or discard the event when cancelled. After that, when an employee node finishes the task, it should publish a response event. The event is annotated with the serial number of the corresponding resource pool. When the event service delivers the event to the requesting node, the pooling service will close the resource pool specified by the serial number, and notify the response event to the agent. Upon closing, the resource pool will send "cancel" messages to all the other employee nodes.

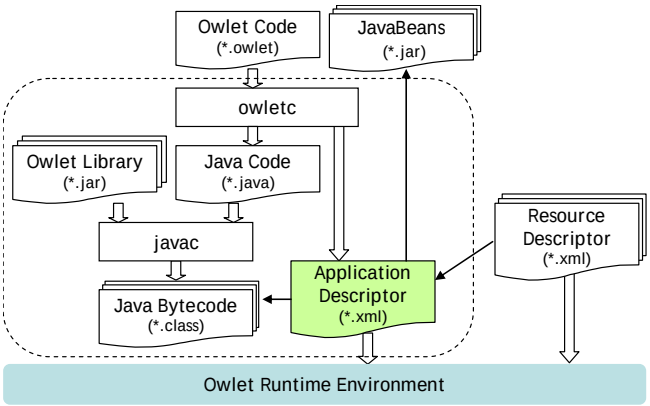

Figure 12. The process of developing an Owlet application.

```
community FileSharing;
schema {
  Piece {
    int id;
    int offset;
    int length;
    string digest;
    string state;
  }

  File {
    string name;
    string path;
    string digest;
    int length;
    int blockSize;
    Piece* pieces;
  }
} // end schema
```

(a) Resource schema definitions

```
role Provider {

  local helper java:beans.FileHelper;

  when ( self: share(File file) ) {

    converse (file.digest) {

      when ( Requester r: request(int* needs) ) {
        Piece* avails = Piece* ();
        enum n: needs do {
          avails = avails + file.pieces[id==n && state=="solid"][0];
        }

        if (avails) // not empty
          provide(avails.id);
      }

      when ( Requester r: request(self, Piece need) ) {
        Piece p = file.pieces[id==need.id][0];
        if (p.state =="solid" && p.digest == need.digest) {
          converse (p.digest) {
            string url = (string) helper.getURL(p);
            provide(url);
            helper.sharePiece(p, url, 20000);
            conclude;
          }
        }
      }
    } // end converse of line 7
  } // end when of line 5
}
```

(b) Provider role

```
role Requester {

  local helper java:beans.FileHelper;

  when ( self: take(File file) ) {
    file = (File) helper.validate(file);

    converse (file.digest) {
      request();

      when ( self: silent(10000) ),
      when ( self: request() ) {
        Piece* pending = file.pieces[state=="empty" || state=="req"];
        if (! pending) {
          join Provider; share(file); conclude;
        } else {
          Piece* needs = file.pieces[state=="empty"];
          if (needs) // not empty
            request(needs.id);
        }
      }

      when ( Provider p: provide(int* avails) ) {
        synchronized (file) {
          enum a: avails do {
            Piece* needs = file.pieces[id == a && state=="empty"];
            if (needs) { // not empty
              Piece need = needs[0];
              need.state = "req";
              converse (need.digest) {
                when (p: provide(string url)) {
                  string path = file.path;
                  if ((bool) helper.fetchPiece(need, url, path)) {
                    need.state = "solid";
                    join Provider; share(file);
                  } else need.state = "empty";
                  request();
                  conclude;
                }

                when ( self: raise(string exception, string message)) {
                  need.state = "empty"; conclude;
                }
              }
              request(p, need);
              break;
            } // end if of line 28
          } // end enum of line 26
        } // end synchronized of line 25
      } // end when of line 24
    } // end converse of line 8
  } // end when of line 5
}
```

(c) Requester role

Figure 13. Owlet program of a peer-to-peer file sharing application.

However, it is possible that none of the employee nodes can survive to publish a response event due to network failures and etc. To detect this situation, the pooling service uses a heartbeat timer to check the conditions of all its employee nodes. If an employee node failed to send a heartbeat message to the resource pool for some threshold time interval, it is removed from the pool. When a pool runs out of employee nodes, a built-in fail event will be triggered for delegating the processing to the programmer.

3.4 Deployment

Besides supporting various language constructs for specifying resource aggregation logic, the Owlet runtime environment also facilitates the deploying and managing of applications for large-scale distributed systems.

XML descriptors are used to deploy applications to distributed nodes. Figure 12 shows the process of developing an Owlet application. First, the resource aggregation logic and the business logic are specified by the developers in an Owlet program and JavaBeans components respectively. The Owlet program is compiled into two parts: its corresponding Java classes and an XML descriptor file for the application. The descriptor contains metadata about the application, e.g. application name, version, codebase, and various configuration parameters. Then, the developers will package the Java classes and JavaBeans components and put them at some network accessible locations, e.g. via FTP/HTTP protocols. The codebase of the descriptor file is set to the URL of the packages. Configuration parameters for the services and overlay networks are set in the descriptor file, e.g. provisioning threshold and timeout, transport addresses of the well-known peers in the overlay network, and etc. Finally, the descriptor file is released to network accessible locations.

The URL of an application descriptor is used to launch the application in a node, which will join the role whose name is specified as a fragment of the URL, e.g. the following URL identifies the `Provider` role of the file sharing application.

`http://owlet-code.sourceforge.net/FileSharing.xml#Provider`

The Owlet runtime environment will parse the descriptor file, download the Java packages, and join the specified role. A Web console and telnet console can be used to provide URLs to the Owlet runtime environment.

Resource descriptor files can also be used to launch an application around particular resources. A resource descriptor contains two parts: a URL referencing the role to join, and an event that will be publishing once joining the role. For example, in the file sharing application described in the next section, a resource descriptor will reference the `Requester` role, and a `take` event which contains the metadata of the file to download.

4. Experiments

In this section, we first demonstrate that the separation of resource aggregation logic and business logic can significantly ease the construction of large-scale distributed systems by building a peer-to-peer file sharing application in Owlet. Then, we illustrate that Owlet programs can achieve a high level of fault tolerance

and a certain degree of load balancing among networked nodes when individual resources are fairly dynamic by using an Owlet distributed prime number checking program.

4.1 Constructing a peer-to-peer file sharing application

Peer-to-peer file sharing is a typical large-scale distributed application that uses the idle bandwidth and storage of other nodes to facilitate file transfer in the Internet. It has the advantage of scalability even when a large number of nodes are downloading files from one node, because each requesting node will provide a trunk of the file for other requesting nodes when possible. Files are usually split into small pieces, so that different pieces can be fetched from different nodes.

When building such an application in Owlet, we first separate it into two parts: 1) the resource aggregation logic that locates the node providing certain pieces, and 2) the business logic that transfers pieces from the located nodes.

The first part is the essence of the peer-to-peer file sharing application, which can be elegantly specified using Owlet, as shown in Figure 13. The metadata of files and pieces used in the application are defined in Figure 13 (a). For instance, the metadata of a file include: the name of the file (`name`), the path to store the file (`path`), the MD5 digest of the file (`digest`), the length of the file (`length`), the size of pieces (`blockSize`) and a list of pieces (`pieces`). The metadata of a piece include: the number of the piece (`id`), the offset of the piece in the file (`offset`), the length of the piece (`length`), the MD5 digest of the piece (`digest`) and the downloading state of the piece (`state`). The state of a piece can be solid, empty or requesting.

Agents are characterized into two roles: `Provider` and `Requester`. Their rules are specified in Figure 13 (b) and (c) respectively. A provider agent provides solid pieces for requester agents. When the user issues a "share" event with the metadata of a file, the rule at lines 5-31 in Figure 13 (b) will be triggered. It will create a conversation by using the digest of the file as the correlation token. The conversation has two enclosing rules. The rule at lines 9-17 in Figure 13 (b) will be triggered when a requester agent publishes a "request" event with a list of piece numbers that it needs. According to this rule, the provider agent will select all available solid pieces the requester needs, and publish a "provide" event containing a list of available piece numbers. The rule at lines 19-29 in Figure 13 (b) will be triggered when a requester agent explicitly publishes a "request" event to this agent asking for a piece. According to this rule, the provider agent checks the state of the piece, and creates a new conversation by using the digest of the piece as the correlation token. In this sub-conversation, it publishes a "provide" event to tell the requester agent the URL to fetch the piece.

A requester agent request pieces from provider agents. And once it has a solid piece, it will join the provider role to provide it to other requesters. When the user issues a "take" event with the metadata of a file, the rule at lines 5-53 in Figure 13 (c) will be triggered. It will first validate the metadata of the file to check the state of all the pieces, and then create a conversation by using the digest of the file as the correlation token. The conversation has two enclosing rules. The rule at lines 12-22 in Figure 13 (c) will be triggered when the agent publishes a "request" event, or a built-in "silent" event. The silent event is published by the conversation when there is no event received or published in the specified time interval. According to the rule, if the agent has empty pieces to download, it will publish a "request" event with a list of the piece numbers. If there are no pending pieces, it will join the Provider role to share the file and conclude the conversation. The rule at lines 24-52 in Figure 13 (c) will be triggered when a provider agent publishes a "provide" event with a list of available piece numbers. According to the rule, it will find the first empty piece which is also in the available list, and set the state of the piece to "requesting". Note that the access to the variable `file` is synchronized to make sure an empty piece will be fetched from one provider only. Then new conversation is created by using the digest of the piece as the correlation token, and a "request" event is published explicitly to the provider to ask for the chosen piece. The sub-conversation has two enclosing rules. The rule at lines 33-41 in Figure 13 (c) will be triggered when the provider publishes a "provide" event with the URL to fetch the piece. If the download succeeds, the agent will join the Provider role to share the piece, otherwise the piece is set to empty state. The rule at lines 43-45 in Figure 13 (c) will be triggered when the agent publishes a built-in "raise" event to report exceptions in the conversation.

The business logic for transferring pieces is provided by a JavaBeans component, which is straightforward to implement. For instance, the component `helper` provides four methods. A provider agent uses the `getURL(piece)` method to get a network accessible location for a piece, and uses `sharePiece(piece, url, timeout)` method to share the piece at the specified location. A requester agent uses the `validate(file)` method to check the pieces of the file (e.g. MD5 digest) and the `fetchPiece(piece, url, path)` method to download the piece from the location and save it to the path.

The Owlet program of the peer-to-peer file sharing application is about 140 lines, which clearly separates the resource aggregation logic and business logic to facilitate the constructing of large-scale distributed systems. The application can be easily deployed and executed without any central control.

We have also rewritten the interactions of a distributed Web page crawler application using Owlet. It originally has 11,000 lines of Java code, in which about 3,000 lines are used to specify the interactions using TCP sockets. Its corresponding Owlet program is about 160 lines, and the rest of its Java code are used as JavaBeans components for the business logic.

4.2 Experimenting fault tolerance and load balancing

Although the Internet can provide a large amount of resources, the interest and behavior of these resources may greatly vary throughout the lifetime of a software system. Resources can spontaneously turn busy, or leave the network without any notification. To comprehensively utilize these resources to build large-scale distributed systems, Owlet provide language constructs that employ multiple redundant resources concurrently for a single task.

A distributed prime number checking program is built using Owlet. To reflect the autonomous and dynamic nature of resources on the Internet, we inject random behaviors into the checking program so that about 40% of the checking processes will fail. We deploy the program to 64 nodes[4] and submit 1000 prime numbers[5] with different redundancy settings at each run. The tasks are submitted at different intervals, and the resources

[4] These nodes are Xen virtual machines running on 16 PowerLeader PR2510D hosts with dual Intel Xeon E5335 quad core processors and 4GB memory. Each Xen VM is exclusively configured with two cores and 512MB memory, and runs Debian GNU/Linux 4.0r3 (x86) with SUN JDK 1.6.0

[5] These prime numbers are from 200000000000027 to 20000000003249, and each of them needs about 10 seconds to check on a single core.

are scheduled with or without sorting by their QoS properties after the provisioning.

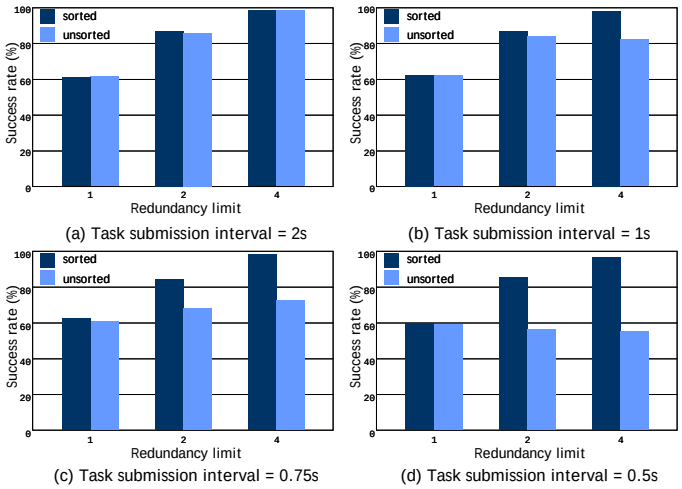

Figure 14. Relationship between the success rates and redundancy limits at different configurations.

Figure 14 shows the relationship between the success rates and the redundancy limit in the Owlet program. When the resources are sorted by their QoS properties before scheduling, the success rates increase as expected when using larger redundancy limits in the Owlet program for all the different task submission intervals. However, when the resources are not sorted before scheduling, the success rates decrease significantly for smaller task submission intervals in comparison to the sorted ones. Because smaller task submission intervals will increase system loads. The sorting provides a certain degree of load balancing among the nodes. When the sorting is absent, some nodes may be overloaded. And they do not have enough computing power to finish the task or even send the heartbeat messages in time. Moreover, when the redundancy limit increases, there will be more redundant tasks submitted to the system, which will worsen the success rates.

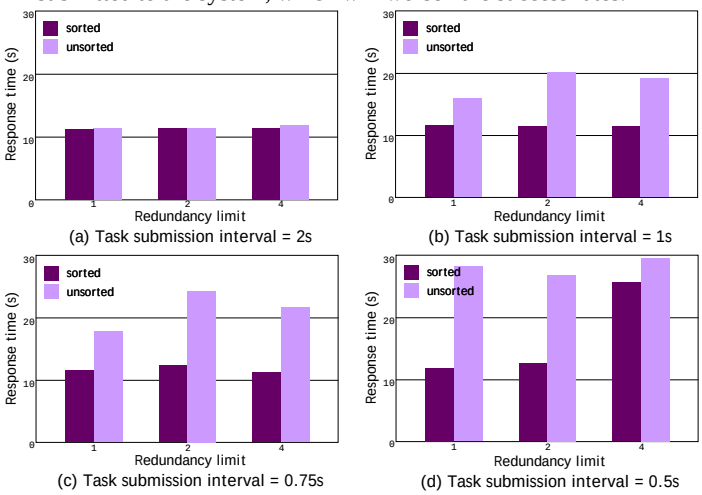

Figure 15. Relationship between response time and redundancy limits at different configurations.

Figure 15 shows the relations between the average response time of the successful tasks and the redundancy limit. The response time with sorting of the nodes before scheduling is more stable than that without sorting. Except in Figure 15 (c) which has the smallest task submission interval, when the redundancy limit is set to 4, the response time increases significantly even with sorting. Because there are 2 tasks submitted a second, each will employ 4 nodes, and a task will take at least 10 seconds to finish. So it requires at least 80 nodes to adequately process the tasks, which exceeds the actual number of nodes in the experiments. Almost all the nodes will be overloaded in this situation, so the response time spikes. However, this will hardly happen when we are building applications in the Internet, where resources are usually abundant.

As shown by the experiments, with the help of the runtime environment, Owlet programs can easily achieve a high level of fault tolerance and a certain degree of load balancing among networked nodes when individual resources are fairly unstable in large-scale distributed systems.

5. Related Work

Event-based approaches have been used to provide asynchrony and decoupling in distributed systems. For example, SEDA [23] is a staged event-driven architecture that supports massive concurrency demands and simplifies the construction of well-conditioned services. Mace [16] is a language extension based on state transition on events that provides a unified framework for networking and event handling, while programming components in a controlled and structured manner. Basically, they focus on the utilization of resources, not covering how to collect and organize resources in large-scale distributed systems. Owlet uses event publish/subscribe interaction scheme as the communication primitive that can elegantly specify resource aggregation logic.

Languages for specifying interaction protocols have been studied in Web services. WS-BPEL [2][19] is a service composition language that provides a set of message exchange primitives and concurrency constructs to specify interactions among services in a business process. XL [11] is a service language that provides high level and declarative constructs for building Web services, e.g. the retry of failed services. GPSL [7] is a service language that integrates messaging, concurrency, and XML data manipulation cohesively. Because their communication primitives are based on point-to-point message passing, programmers need to bind services manually at deployment and explicitly handle fault tolerance and load balancing when using services on the Internet.

Interaction-oriented programming emerges as new paradigm in multi-agent systems based on interacting agents, active objects, and active wrappers of legacy components [13]. For example, OWL-P [8] provides primitives such as roles, the messages exchanged between them, and declarative rules describing the effects of messages in terms of commitments. IOM/T [9] is an interaction description language which has correspondences with AUML sequence diagrams. However, they focus on describing interactions among a prior determined set of agents, i.e. they do not have language support for discovering and organizing other agents. SLABSp [22] uses scenario rules to specify interactions among agents in patterns of action sequences, but it has little support for message correlation, fault tolerance and load balancing. PIAX [15] is a framework that integrates mobile agent based messaging and peer-to-peer discovery mechanisms transparently with application layer multicasts. Channeled Multicast [4] uses themed multicast to facilitate the specification of interaction protocols, while Owlet provides a more expressive means by using content-based publish/subscribe interaction scheme.

There are also many distributed computing middleware aimed at building large-scale distributed systems. JXTA [12] technology is a network programming and computing platform based on Java

that provide a framework for building peer-to-peer networking applications. BIONIC [1] is an open infrastructure for network computing that provides a framework for building large-scale distributed applications. ProActive [5] is an open source Java library aiming to simplify the programming of multithreaded, parallel, and distributed applications. They provide frameworks and libraries for building distributed systems which conform to certain schemes.

6. Conclusion and Future Work

In this paper, we present Owlet, an interaction language based on event publish/subscribe communication primitive to specify the interactions among distributed nodes to collect, organize and comprehensively utilize of the resources on the Internet. It abstracts the resources as agents, and the interactions are described in terms of different roles of the agents. By using peer-to-peer overlay network and content-based publish/subscribe event service as the enabling technologies, Owlet provides the language constructs that facilitate the construction of large-scale distributed systems by separating its resource aggregation logic and business logic.

We have implemented an Owlet compiler and its runtime environment. Experiments show that large-scale distributed applications can be elegantly constructed in Owlet, and a high level of fault tolerance and a certain degree of load balancing among networked nodes can be achieved.

It is considered to incorporate more resource pooling mechanisms as described in [24] to provide more reliable access to resources on the Internet. We are also investigating the trust and incentive mechanisms in building distributed applications to provide better quality of services in large-scale distributed system.

References

[1] Anderson, D.P. BOINC: A System for Public-Resource Computing and Storage. In *Proc. 5th Int'l Workshop on Grid Computing (GRID'04)*, IEEE CS Press, pp. 4-10, 2004.

[2] Alves, A., Arkin, A. and et al. Web Services Business Process Execution Language Version 2.0, *OASIS Standard*, 2007.

[3] Androutsellis-Theotokis, S. and Spinellis, D. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4): 335-371, 2004.

[4] Busetta, P., Dona, A. and Nori, M. Channeled Multicast for Group Communications. In *Proc. 1st Int'l Joint Conf. on Autonomous Agents and MultiAgent Systems (AAMAS'02)*, ACM Press, pp. 1280-1287, 2002.

[5] Caromel, D., di Costanzo, A. and Mathieu, C. Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Computing*, Elsevier, 33(4-5): 275-288, 2007.

[6] Clark, J. and DeRose, S. XML Path Language (XPath) Version 1.0, *W3C Recommendation*, 1999.

[7] Cooney, D., Dumas, M. and Roe, P. GPSL: a programming language for service implementation. In *Proc. 9th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE'6)*, Springer-Verlag, LNCS 3922: 3-17, 2006.

[8] Desai, N., Mallya, A.U., Chopra, A.K. and Singh, M.P. Interaction protocols as design abstractions for business processes. *IEEE Trans. on Software Engineering*, 31(12): 1015-1027, 2005.

[9] Doi, T., Tahara, Y. and Honiden, S. IOM/T: an interaction description language for multi-agent systems. In *Proc. 4th Int'l Joint Conf. on Autonomous Agent and Multi-Agent Systems (AAMAS'05)*, ACM Press, pp. 778-785, 2005.

[10] Eugster, P., Felber, P., Guerraoui, R. and Kermarrec, A. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2): 114–131, 2003.

[11] Florescu, D., Grünhagen, A. and Kossmann, D. XL: an XML programming language for Web service specification and composition. *Computer Networks*, Elsevier, 42(5): 641-660, 2003.

[12] Gong, L. JXTA: a network programming environment. *IEEE Internet Computing*, 5(3): 88-95, 2001.

[13] Huhns, M.N. Interaction-Oriented Programming. In *Proc. 1st Int'l Workshop on Agent-Oriented Software Engineering (AOSE'00)*, Springer-Verlag, LNCS 1957: 29-44, 2000.

[14] Jennings, N.R. An agent-based approach for building complex software systems. *Comm. ACM*, 44(4): 35-41, 2001.

[15] Kaneko, Y., Harumoto, K. and et al. A Location-Based Peer-to-Peer Network for Context-Aware Services in a Ubiquitous Environment. In *Proc. 2005 IEEE/IPSJ Int'l Symp. on Applications and the Internet (SAINT'05) Workshops*, IEEE CS Press, pp. 208-211, 2005.

[16] Killian, C., Anderson, J.W., Braud, R., Jhala, R. and Vahdat, A. Mace: language support for building distributed systems. In *Proc. 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'07)*, pp. 179-188, 2007.

[17] Li, D., Lu, X. and Wu, J. FISSIONE: a scalable constant degree and low congestion DHT scheme based on Kautz graphs. In *Proc. 24th Annual Joint Conf. of the IEEE Computer and Comm. Societies (INFOCOMM'05)*, IEEE CS Press, pp. 1677-1688, 2005.

[18] Lu, X., Wang, H. and Wang, J. Internet-based virtual computing environment (iVCE): concepts and architecture. *Science in China (Series F)*, Springer-Verlag, 49(6): 681-701, 2006.

[19] Paurobally, S. and Jennings, N.R. Protocol engineering for web services conversations. *Engineering Applications of Artificial Intelligence*, Elsevier B.V., 18(2):237-254, 2005.

[20] Platon, E., Mamei, M., Sabouret, N., Honiden, S. and Parunak, H.V.D. Mechanisms for environments in multi-agent systems: survey and opportunities. *Autonomous Agents and Multi-Agent Systems*, Springer-Verlag, 14(1): 31-47, 2007.

[21] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. and Balakrishnan, H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. 2001 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM'01)*, ACM Press, pp. 149-160, 2001.

[22] Wang, J., Shen, R. and Zhu, H. Agent-oriented programming based on SLABS. In *Proc. 29th Annual Int'l Computer Software and Applications Conf. (COMPSAC'05)*, IEEE CS Press, pp. 127-132, 2005.

[23] Welsh, M., Culler, D. and Brewer, E. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'01)*, ACM Press, pp. 230-243, 2001.

[24] Zheng, Z. and Lyu, M.R. WS-DREAM: A Distributed Reliability Assessment Mechanism for Web Services. In *Proc. 38th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN'08)*, IEEE CS Press, pp. 392-397, 2008.

[25] Zhu, Y. and Hu, Y. Ferry: a P2P-based architecture for content-based publish/subscribe services. *IEEE Trans. on Parallel and Distributed Systems*, 18(5): 672-685, 2007.