Peer: an Architectural Pattern[*]

M. Amoretti, M. Reggiani, F. Zanichelli, G. Conte
Distributed Systems Group
Dipartimento di Ingegneria dell'Informazione
Parco Area delle Scienze, 181A, I-43100, Parma, Italy
{amoretti, reggiani, mczane, conte}@ce.unipr.it

Abstract

Virtual Organizations (VOs) are transitory communities made of individuals and institutions coupled together by advanced communications technologies. Today's VOs are more and more demanding in terms of efficiency and flexibility of the technological infrastructure. Envisioning VOs in which all participants are both resource providers and consumers, in a peer-to-peer fashion, seems to be an appropriate solution.

This paper proposes a structure for programmed elements, *i.e.* Peers, sharing resources through a peer-to-peer network. Publishing and discovery of resource provision services are managed by the routing and communication modules. The first one creates publication/query messages and computes their destination, which is then used by the communication service to transport the message to the destination Peer.

Intent

The Peer pattern enables Virtual Organizations whose participants collaborate to deploy, publish, discover, and consume heterogeneous resources (CPU, storage, bandwidth, applications).

Example

Community support systems is gaining more and more attention in application areas ranging from leisure and customer support to knowledge management. ICT, in particular, is becoming of paramount importance for the so-called Informal Learning, which includes all the activities taking place outside classes and course that aim at understanding, creation of knowledge, and skills acquisition.

Several universities around the world have already deployed or are experimenting solutions for course content distribution over their TCP/IP wired/wireless networks and the Internet. In a campus-wide e-learning scenario, class lectures, seminars, lab sessions are made available in digital format to be downloaded and viewed either off or on-campus.

Most universities have typically a limited amount of resources, which are contended by many students. At the same time, it is very likely that in these same institutions there are hundreds of workstations that remain idle most of the time. In this context Grid architectures [10, 15] can minimize this waste of resources, augmenting the processing power available to students. Unfortunately, most of current Grid architectures rely on a rigid client/server approach which has many drawbacks. A few members of the university, mostly professors, are service providers publishing their services within centralized indexes, while students are usually service consumers.

Thus, in traditional Grids a considerable wealth, namely the whole amount of student resources (CPU cycles, storage, etc.), remains unused. Moreover, the centralized index approach

[*]Copyright 2005 by M. Amoretti, M. Reggiani, F. Zanichelli, G. Conte. Permission is granted to copy for the PLoP 2005 conference. All other rights reserved.

for service publishing and discovery (*e.g.* a UDDI registry) is neither efficient nor robust. When the indexing service has to sustain a demanding load, its response time may become too high. Finally, it constitutes a single point of failure for the whole system.

Context

Resource sharing in Virtual Organizations.

Problem

Virtual Organizations need efficient workload distribution and high resource availability. These features are not fully guaranteed by the client/server approach, because user resources may remain unused. Moreover, providers' resources are usually published/searched through a centralized repository or broker, introducing single points of failure and scalability issues. A better solution should consider the following *forces*:

- All participants must be active contributors to the distributed functionalities of the system, without rigid distinction between service providers and service consumers.
- Robust resource sharing mechanisms must be provided, avoiding single points of failure.
- Security mechanisms (key management, authentication, admission control, authorization, privacy) should be provided, not limited to user-to-service interactions but also for service-to-service interactions.
- Support for participant ranking, based on their capabilities and trustiness, should be granted.

Solution

The Peer pattern is a viable solution to this problem. In a Peer-based distributed system, *i.e.* peer-to-peer system, all nodes have the same structure and are not only resource consumers but also resource providers. The nature and availability of resources that Peers can share is multifold: storage, CPU cycles, content, real-time sensor data, applications.

A dynamic set of Peers which have a common set of interestes, and have agreed upon a common set of policies, is called peergroup. Malicious behaviors (such as message flooding, unauthorized service usage/provision, etc.) should be prevented by the *Security Manager* of each peergroup the Peer belongs to.

The *User Interface* is the module which is responsible for user-to-Peer interactions. External events (generated by the user or by other nodes) affect the Peer state, which can be accessed through the *State Manager*. Each resource is exposed through a *Resource Provision Service*, whose interface is published in the peer-to-peer network. Messages are propagated according to policies established by the *Router* and *Message Handler* modules. Finally, each Peer manages external requests to its resources using the *Scheduler*, whose decisions are based on information provided by the *Resource Monitor*.

Participants

In the following we describe the Peer modules, whose CRC cards are presented in figure 1.

- The *Message Handler* allows Peers to create a virtual network overlay on top of the existing physical network infrastructure. This module supports different transport protocols allowing message transmissions, potentially traversing firewalls or NATs.

Message Handler	
Responsibilities	Interactions
Provides a reliable communication mechanism.	*Router* *Security Manager* *Resource Provision Service* *State Manager*

Scheduler	
Responsibilities	Interactions
Manages the execution of task requests	*Resource Provision Service* *Resource Monitor* *User Interface* *Router* *Security Manager*

Resource Provision Service	
Responsibilities	Interactions
Exposes methods which allow Peers to interact with a resource	*User Interface* *Scheduler* *Message Handler*

Security Manager	
Responsibilities	Interactions
Is responsible for protecting shared resources	*Scheduler* *Message Handler* *Router*

Resource Monitor	
Responsibilities	Interactions
Gathers and reports information about local resources	*User Interface* *Scheduler*

Router	
Responsibilities	Interactions
Defines the rules for propagating messages	*User Interface* *Message Handler* *Scheduler* *Security Manager*

User Interface	
Responsibilities	Interactions
Allows the user to interact with the system of Peers	*Router* *Resource Provision Service* *Resource Monitor* *Scheduler* *State Manager*

State Manager	
Responsibilities	Interactions
Provides facilities for checking and changing the state of the Peer	*Message Handler* *User Interface*

Figure 1: CRC cards describing the Peer modules.

- A pool of *Resource Provision Service*s implements resource management mechanisms for local and remote control of Peer's resources (*e.g.* computational power, storage, multimedia objects, sensor data). The same resource may be used in several ways, depending on sharing restrictions (*i.e.* when, where, and what can be done). These constraints can dynamically change, based on the nature of permitted access, and on participants to whom access is allowed.

- On the other hand, Peer's resources provide enquiry mechanisms allowing analysis of their structure, state, and capabilities. The *Resource Monitor* relies on these facilities to gather and report information about local resources. The Resource Monitor is responsible for providing a list of available resources, maintaining related information, identifying and reporting failures.

- The *User Interface* is the entry point for users to interact with the system of Peers. The module is responsible for managing user commands, which are opportunely translated in local control operations or messages to be propagated in the network. The User Interface allows to query the Resource Monitor and to configure local resources, to express complex queries for remote resources and to operate on them.

- The *Router* defines the rules for addressing, filtering, sending, and receiving messages. Most of the Peer operations, such as publication, search and delivery of resources, rely on the Router.

- The *Scheduler* is responsible for managing task execution requests, based on information provided by the Resource Monitor. Since many Peers can provide the same resources, their Scheduler modules must be able to cooperate for workload distribution.

- The *Security Manager* is mainly responsible for protecting shared resources. It relies on mechanisms to safeguard integrity and authenticity of data, to ensure privacy and confidentiality in communications, and to provide means for user authentication and authorization. Moreover, the Security Manager allows Peers to establish their identity within a group. In general, the Security Manager should define the capabilities of the Peer within a peergroup, *i.e.*, in increasing order of importance (and trustworthiness required): message forwarding, resource discovery, resource publishing and delivery, group monitoring, group management (*e.g.* voting for changing the rank of another group member), subgroup creation.

- The *State Manager* provides facilities to check and change the Peer state. *E.g.* in a peergroup the Peer could be a supernode (with all its capabilities activated), while in another peergroup it could be a leaf node (unable to perform certain actions). Peer state changes can be user-driven, or dependent on the configuration of the peergroup.

Some of these modules rely on distributed algorithms, *e.g.* the resource discovery functionality is realized through the cooperation of Routers belonging to many distinct Peers.

Structure

Figure 2 shows a common Peer-based system, emphasizing intra- and inter-Peer module interactions.

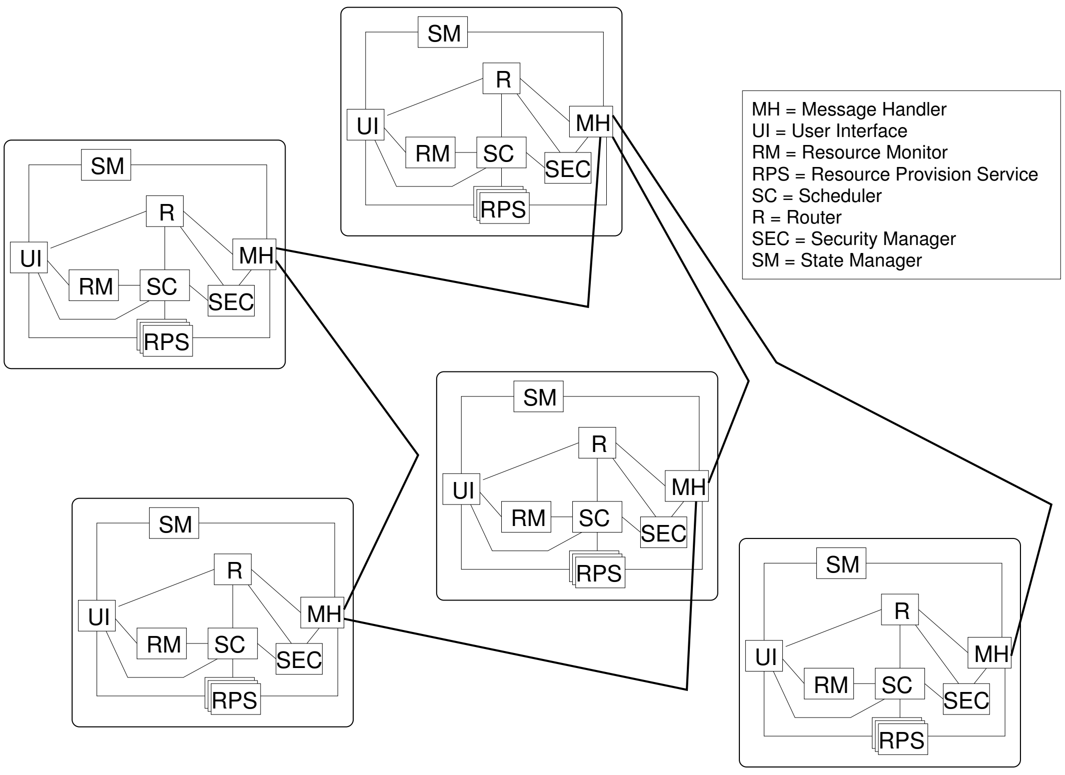

Figure 2: Example of Peer-based system. For each Peer, interactions among internal modules are illustrated.

Connections among participants belonging to the same node are instead illustrated in figure 3; Peer services are represented in terms of abstract classes.

The User Interface is aware of one or more Resource Provision Services, either local or remote. Moreover, it must be able to query the Resource Monitor and the State Manager, to collect information about the system and to provide it to the user. Finally, the User Interface uses the Router when it publishes its Resource Provision Services or searches fors remote ones.

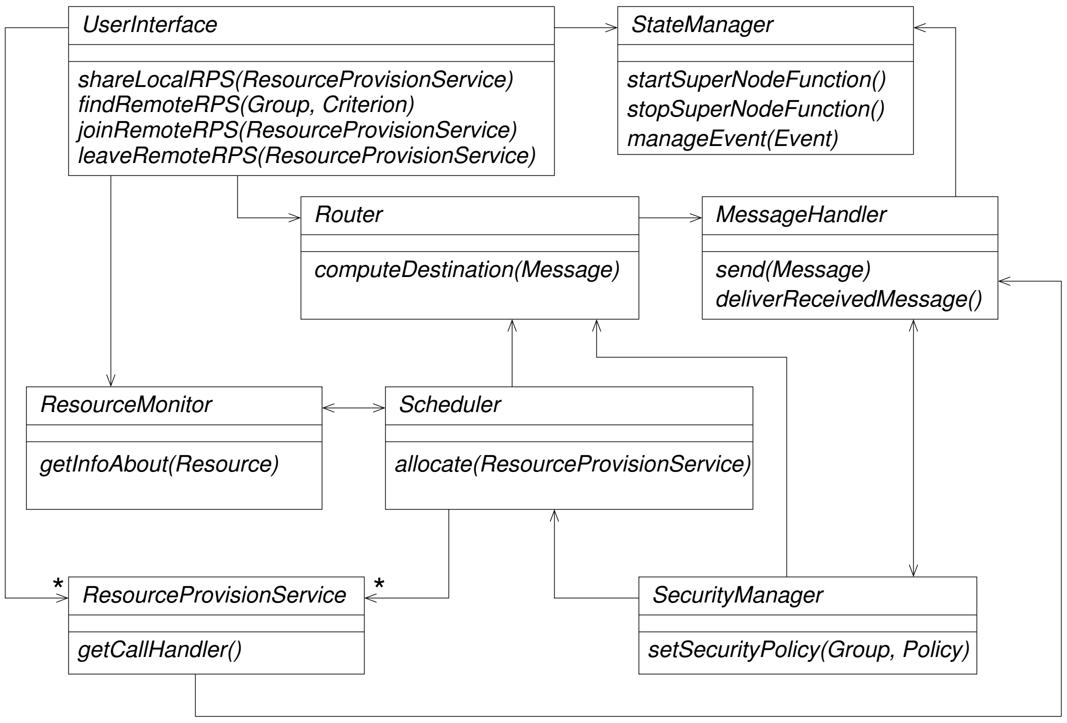

Figure 3: Class diagram of the Peer.

The State Manager can modify the state of the Peer as a consequence of a user command received from the User Interface, or after the delivery of an event by the Message Handler. For this reason the State Manager is referred by both the User Interface and the Message Handler.

The Router computes the destination of messages constructed by the User Interface. Messages can be distributed using the supported transport protocols accessed through one Message Handler.

When a remote request for service provision is received, the Scheduler queries the Resource Monitor for available resources. On positive answer, the Scheduler activates the appropriate Resource Provision Service based on the call handler.

The Security Manager interacts with the Scheduler, Router, and Message Handler, to modify the behaviour of the peer in the current peergroup.

Dynamics

Scenario I: State Change

As illustrated in figure 4, the Message Handler delivers event messages to the State Manager, which performs transitions according to that messages. The User Interface inspects the State Manager to know which actions are allowed to the user, based on the state of the Peer.

Scenario II: Resource Sharing/Discovery

Figure 5 illustrates the process of publishing and searching a resource.

- When a resource is published, the interface of its Resource Provision Service, along with other information such as security requirements, is passed to the Router. On resource search, (eventually complex) queries are passed to the Router.

- The Router creates the message(s) and computes the destination(s). Many different algorithms can be applied, as explained in the Implementation section.

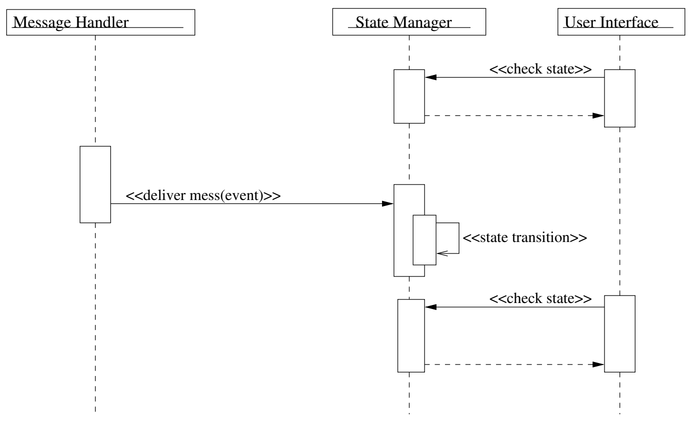

Figure 4: How the Peer state changes (scenario I).

- The Router sends the message(s) to the computed destination(s).

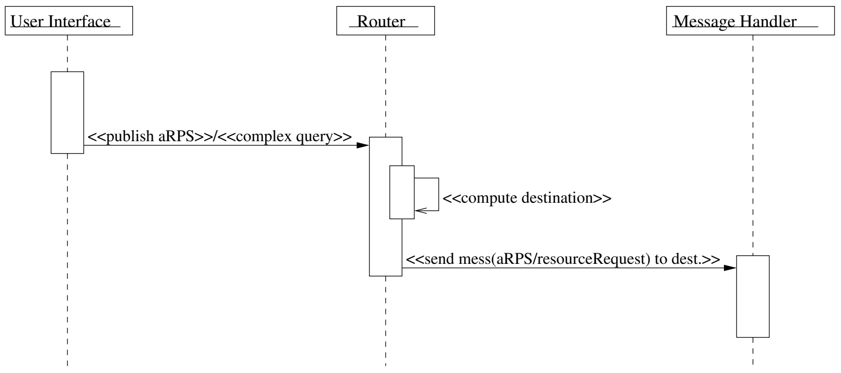
User Interface
Router
Message Handler
<<publish aRPS>>/<<complex query>>
<<compute destination>>
<<send mess(aRPS/resourceRequest) to dest.>>

Figure 5: How the Peer performs sharing/discovery of resources (scenario II).

Often, the Peer acts just as a message propagator. In this case, it receives a message from another Peer, computes the new destination(s), and sends the message accordingly.

Scenario III: Resource Provision

The most interesting scenario is resource delivery, involving multiple Peer's modules (figure 6).

- The Message Handler delivers a message to the Security Manager, communicating resource request and credentials of the requester (another Peer).
- If the requester's credentials are valid, the Security Manager delivers the resource request to the Scheduler.
- The Scheduler interacts with the Resource Monitor to check the availability of the requested resource.
- If the requested resource is available, the Scheduler invokes its allocation on the corresponding Resource Provision Service.
- If the resource is not available, or by its nature can be provided concurrently by many Peers, the Scheduler starts searching for remote resources, transparently to the user. The Router computes the destination(s) and sends request message(s).
- In the meantime, the Resource Provision Service provides the available local resource to the requester, through the Message Handler.

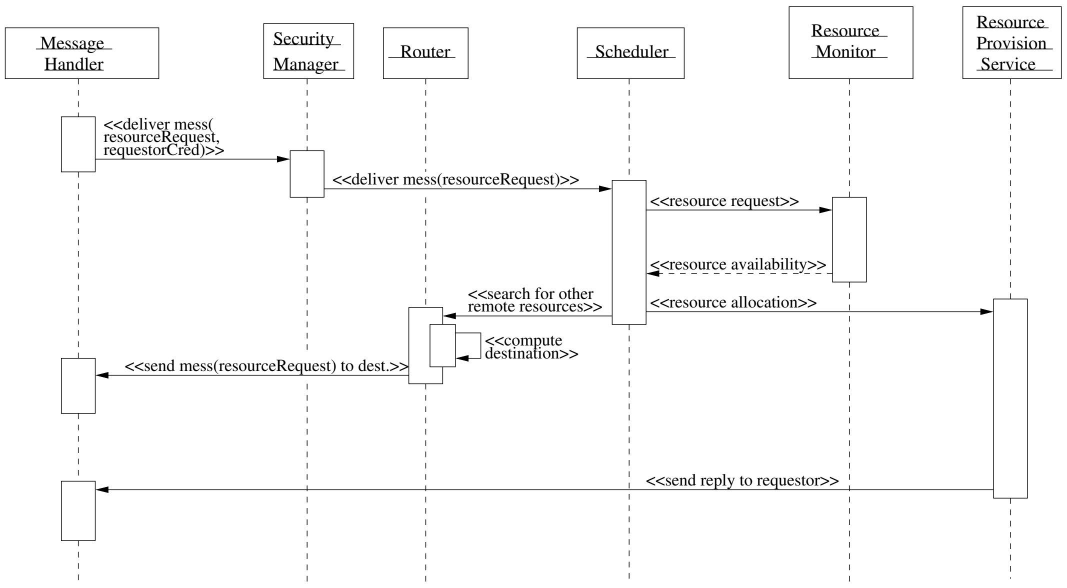
Message Handler
Security Manager
Router
Scheduler
Resource Monitor
Resource Provision Service
<<deliver mess(resourceRequest, requestorCred)>>
<<deliver mess(resourceRequest)>>
<<resource request>>
<<resource availability>>
<<search for other remote resources>>
<<resource allocation>>
<<compute destination>>
<<send mess(resourceRequest) to dest.>>
<<send reply to requestor>>

Figure 6: How the Peer provides its resources (scenario III).

Implementation

In this section we illustrate issues to be aware of, when implementing the Peer pattern's modules.

Implementing the Message Handler

Peers should be able to communicate independently of their location in the underlying physical network. To this purpose, virtual network abstractions are necessary, composed of a logical addressing model, a resolver performing all binding operations required in a distributed system, and virtual communication channels. To this purpose, Remoting Patterns may be helpful [21]. In the following we focus on the most important and used technologies.

An example of basic reliable communication mechanism is the Remote Procedure Call (RPC) [4]. RPCs are special function calls embedded within the client portion of the application program. As they are embedded, RPCs do not stand alone as a discreet middleware layer. When the client program is compiled, the compiler creates a local stub for the client portion and another stub for the server portion of the application. These stubs are invoked when the application requires a remote function and typically support synchronous calls between clients and servers. By using RPCs, the complexity involved in the development of distributed processing is reduced, as a common semantics for a remote call is used, whether or not the client and server are distributed.

An RPC-based solution is Java Remote Method Invocation (Java RMI) [14], which enables distributed Java-based applications. Java RMI is application-independent, but it is language-dependent.

The .NET Framework [13] is Microsoft's solution for building and running applications which communicate across network boundaries. With respect to its predecessor, the RPC-based Distributed Component Object Model (DCOM), .NET provides developers with a significant number of benefits including a more robust, evidence-based security model, automatic memory management and native Web Services support. Although it supports over 20 different programming languages, .NET is currently implemented only for Windows.

A robust industry standard is CORBA [16], which provides language, vendor, and operating system independence. CORBA fundamental component is the Object Request Broker (ORB) that behaves as a system bus, connecting objects operating in an arbitrary configuration thus ensuring portability, reusability, and interoperability. Service request in CORBA can be either blocking (Synchronous Method Invocation - SMI) or non-blocking (Asynchronous Method Invocation - AMI). The latter allows a CORBA-based system to efficiently activate multiple concurrent actions on remote objects. Moreover, as SMI and AMI share the same object interface, clients can choose between synchronous or asynchronous calls without affecting server implementation.

Enabling interaction with the system

The User Interface, which can be textual or graphical, allows the user to see local resources, to choose which of them sharing in the network, to monitor their state and to set constraints on their usage. Moreover, the User Interface must provide facilities for remote resource discovery and interaction.

Complexity of user queries depends on how Resource Provision Services are described. If a RPS has a WSDL interface [7], then all its functionalities can be retrieved. Unfortunately, a WSDL document is unable to express the semantics of the service, thus preventing important features such as automatic service composition and invocation, as well as execution monitoring. For this reason, OWL-S descriptions [17] are the new trend in service-oriented architectures. In a peer-to-peer context, OWL-S descriptions allow to contextualize the services in an ontology shared among peers. Such complete descriptions are useful to elaborate complex high-level queries, which are translated by the User Interface to sets of atomic messages for the Router, which performs the low-level peer-to-peer search process.

The User Interface must also be able to interact with remote RPSs, based on the information provided by their interface. This is OWL-S's concept of grounding, which is generally consistent with WSDL's concept of binding.

Implementing Resource Provision Services

In the context of the Peer pattern, we assume that shareable resources are all represented as services, *i.e.* network-enabled entities which provide some capabilities. Service-orientation allows to address several issues: the need for standard interfaces, local/remote transparency, adaptation on local OS functionalities, and virtualization, *i.e.* common interface of diverse implementations.

The CORBA framework [16] provides a Concurrency Control Service which allows several clients to coordinate their concurrent accesses to a shared resource, so that the resource's consistent state is not compromised. Unfortunately, the Concurrency Control Service does not define what a resource is. It is up to the developer to define resources and identify situations where concurrent accesses to resources conflict.

For this reason, a better option to implement RPSs is the Web Services Architecture [22]. Web Services provide several advantages, such as: portability, accessibility over firewalls/NATs, adequateness for loosely coupled systems. In fact, Web Services are platform-independent and language-independent, since they use standard XML languages. Most of them use HTTP for transmitting messages, since most of the Internet's proxies and firewalls let HTTP traffic traverse them (unlike CORBA, which usually has trouble with firewalls). Moreover, a client might have no prior knowledge of the Web Service until it actually invokes it. These are major advantages to build Internet-scale applications. Transmitting all data in XML is obviously not as efficient as using a proprietary binary code. Even so, this overhead is usually acceptable for most (non real-time) applications.

Plain Web Services are not very versatile, since they only allow for some very basic forms of service invocation. CORBA, for example, offers programmers a lot of supporting services (such as persistency, notifications, lifecycle management, transactions, etc.). Emerging specifications are currently enhancing Web Services. To provide stateful Web Services, the Web Services Resource Framework (WSRF) [23] has been recently proposed. WSRF defines a generic and open framework for modeling and accessing stateful resources using Web Services. This framework includes mechanisms to describe views on the state, to support state management through properties associated with the Web service, and to describe how these mechanisms are extensible to groups of Web services.

Monitoring and Scheduling resources

The overall topology of the system of Peers is generally unpredictable, as the set of participants changes. Thus, ensuring vertical reliability (*i.e.* reliability of Peers' local resources) and horizontal reliability (*i.e.* reliability of operations involving many Peers) is a major challenge. Automated fault recovery and task resuming can be achieved by redundant execution, use of checkpoints and task migration. These mechanisms must not introduce a large overhead, which would affect task execution performance.

The Resource Monitor could be an ensemble of sub-services, each one referring to a particular resource. Monitoring tools tend to operate at different levels of granularity, with consequent tradeoffs between the quality of the information monitored and the associated overheads. Some tools are able to predict future behavior based on past history. A good survey of fine-grain monitoring tools is proposed in [1]. Most of them are based on the Publisher/Subscriber communication model [5]: whenever the Publisher (the resource) changes state, it sends a notification to all its Subscribers (the sub-services which compose the Resource Monitor). Subscribers in turn retrieve the changed data at their discretion. CORBA proposes two variants of the Publisher/Subscriber communication model: the Event Service and the Notification Service. These solutions strongly decouple Publisher and Subscribers by means of an Event Channel, which acts as a proxy consumer for the real suppliers and as a proxy supplier towards the real consumers. Therefore, the Publisher can perform a non blocking send of data in the Event Channel, while the interested Subscribers can connect to that channel to get the event. Distributed resource monitoring, spanning many Peers, should be based on the exchange, through the Router, of local monitoring information acquired with the previously illustrated techniques.

The Scheduler provides task decomposition and mapping. Task decomposition is the process of dividing a complex task (maybe involving different resources) into smaller parts, which may potentially be executed concurrently by different Peers. Mapping is the mechanism by which tasks are assigned to processes for execution (maybe spanning different Peers). A mapping algorithm should seek to maximize the concurrency of tasks while minimizing the interaction among processes, *i.e.* message exchange between Peers, with the goal of reducing total completion time.

Implementing routing mechanisms for resource sharing and discovery

The efficiency of a message routing algorithm is strictly related to the topology of the Peer network. At a logical level (*i.e.* addressing space) Peers may form unstructured or structured topologies.

The *Centralized Directory Model (CDM)* was made popular by Napster. The resulting architecture is hybrid: community peers connect to a central directory where they publish information about resources they offer for sharing. Upon request from a peer, the central index matches the request with the peer in its directory offering the best answer. The best peer could be the one that is cheapest, fastest, or most available, depending on user needs. There are scalability limits, because more performing servers are required when the number of requests increase. Napster's experience showed that this model is very efficient, while not secure and robust.

A Router based on the *Distributed Indexes and Repositories Model (DIRM)* relies on some peers which act as brokers. When a peer sends a query to a broker, the latter replies with information about matching resources (if any) and pointers to owner peers. Queries are also forwarded to neighbor brokers. This solution requires index replication between brokers, and efficient update mechanisms. OpenNap, MP2P and eDonkey networks are based on this model.

In the *Flooded Requests Model (FRM)*, each request from a peer is broadcasted to directly connected peers, which themselves flood their neighbors, etc. This solution, which is adopted by Gnutella, consumes a lot of network bandwidth, but is efficient in limited communities such as company networks.

The *Selective Queries Model (SQM)* uses a flow control algorithm for query propagation, searching for peers which have the highest probability of fulfilling the resource requirements. The control algorithm is sometimes implemented only by peers with higher bandwidth and process capacity, which are named super-peers or supernodes. This model is used by FastTrack and Gnutella2.

The *Document Routing Model (DRM)*, based on Distributed Hash Tables (DHT), is the state-of-art approach in the field of pure peer-to-peer architectures. Each peer has a randomly assigned ID and has a logically structured list of neighbors. When a resource is published (shared) on such a system, its description is hashed to obtain an ID. Each peer then routes the resource description towards the peer with the ID that is most similar to the description ID. This process is repeated until the nearest peer ID is the current peer's ID. When a peer requests a resource from the P2P system, the request goes to the peer with the ID most similar to the description ID. This process is repeated until a matching resource is found. This approach is used, for example, in the Distributed K-ary Search (DKS) [2] family of protocols.

Securing the system

It is quite evident that providing security to groups of Peers is a difficult problem, because interactions are not just user-to-service, for accessing resources, but also service-to-service on behalf of different Peers, thus requiring delegation of rights. Moreover, the implementation must be broadly available and applicable, *i.e.* standard, well-tested, well-understood protocols are needed. Finally, a number of different policies need to be combined.

The goal of the Security Manager is ensuring that no participant uses an unfair share of the resources. Thus, group membership and resource access generally require policies for:

- key management
- authentication
- admission control
- authorization
- transport security

These policies can rely on two orthogonal approaches for group security: the reputation scheme, and the trust negotiation scheme [25]. In a reputation system, a Peer makes decisions on its own experience and other Peers' recommendations. Even though some form of persistent node identification is required, reputation systems are not considered suitable for critical tasks. This weakness is addressed by the other scheme, which builds trust by exchanging digitally signed certificates. An important standard for certificates is $X.509$ [12], which is used in a variety of contexts.

On the other hand, complicated security mechanisms can adversely affect performance. For this reason the ability to dynamically control security levels based on information known at runtime is a desirable feature.

Managing the State of the Peer

A Peer can be considered a *reactive system* [24] because of its continual interaction (user-driven or transparent) with its environment, which is constituted by other Peers, and execution at a pace determined by that environment. For example, in case of Router based on the Selective Queries Model, a Peer can dynamically change its state and become a supernode, if its bandwidth resources are sufficient. Moreover, in a secured environment, peergroup members could have different ranks, corresponding to the actions they are allowed to perform within the group. Promotions and demotions should be based on the history of Peer's interactions with its environment.

The Finite State Machine (FSM) of the Peer should be explicitly implemented in the State Manager. The FSM represents all the details of a control situation, without being involved in the implementation of the actions. The FSM references the actions but knows them only by name, thus the control mechanism can be changed, in one place (the FSM) without changing the implementations of the actions. Similarly, the actions can be changed without influencing the control mechanism. This is an extremely useful separation of concerns.

Dyson and Anderson have defined a language of State Patterns [9] to deal with the implementation of state-dependent behavior. The central pattern to the language is the State Object, which encapsulates the state of an object in another, separate, object. The Exposed State pattern details when it is necessary to allow external objects to have direct access to the owning object's State Object. Finite state machines can be based on State-Driven Transitions, with the State Object being responsible for the transition to one state to another, or Owner-Driven Transitions, dealing with the alternative approach which is for the owning object to initiate transitions between states. All these patterns are evidently useful to implement the State Manager module of the Peer pattern.

Example Resolved

In a mutable environment, with a large number of students dynamically connecting to the system, the traditional grid approach where services are hosted by always available servers (e.g. the teachers' ones) and users know in advance where to find the required resources is unsatisfactory. Indeed, to provide high levels of scalability, decentralization, and interoperability, students can also be used as service providers, e.g. they can offer computational power, additional storate, etc. This kind of scenario requires to exceed the Client/Server approach of traditional Grids and considers a Peer-to-Peer solution to realize a new conception of Grid.

Moreover, the Peer pattern introduces a distributed approach to resource publishing and discovery, where the overall university members collaborate to realize and maintain a sort of distributed repository. Messages are propagated among Peers according to the rules defined by the Router. The overlay network can be completely decentralized and symmetric, or based on supernodes acting as indexers.

The overall security is guaranteed by the Security Manager that prevents an unfair use of available resources, preserving Peers from different attacks, such as message flooding and malicious service invocations. Security Manager implementation is based on standard, well-tested, well-understood protocols for key management, authentication, admission control, authorization and transport security.

Known Uses

A widespread peer-to-peer application is content distribution, which is based on the coordinated use of personal computers to function as a distributed storage medium by contributing, searching, and obtaining digital content. An almost complete survey on peer-to-peer content distribution technologies is proposed in [3].

Among many interesting peer-to-peer projects, in different application fields, we evaluated those regarding open source peer-to-peer middleware.

JXTA [19], a Sun Microsystems open initiative, is the most supported (and advanced) peer-to-peer middleware currently available. The main effort of JXTA developers is to standardize a common set of protocols which define the minimum required network semantics, allowing peers to form and join an unstructured peer-to-peer network based on supernodes. The middleware, which has both Java and C bindings, includes the following modules: endpoint, resolver and pipe, implementing the Message Handler; meter, implementing a partial Resource Monitor; rendezvous and discovery, implementing the Router; credential and membership, implementing the Security Manager; peer, implementing the State Manager. The other modules of the Peer pattern are considered application-dependent, thus to be built over the middleware.

DKS [2] is a peer-to-peer middleware developed at KTH/Royal Institute of Technology and the Swedish Institute of Computer Science (SICS). It supports scalable Internet-scale Multicast, Broadcast, Name-based Routing, and provides a simple Distributed Hash Table abstraction. The middleware, which is written in Java, includes the following packages: dks_comm, implementing the Message Handler; dks_dht and dks_marshal, implementing the Router; dks_node, implementing a sort of State Manager. Resource provision and security issues are not currently addressed.

P2PS [6] is a middleware developed by the Université Catholique de Louvain (UCL) and CETIC. P2PS offers various primitives for joining/leaving a network, group communication (one-to-one, broadcast and multicast), monitoring, and data location. P2PS implements Tango, an algorithm for constructing structured P2P systems, which appears to be more scalable than logarithmic-degree DHT based networks. The P2PS library is implemented in Oz with the Mozart platform [20], thus one needs the Oz virtual machine to compile and use it. The P2PS library contains the class P2PServices, which allows a peer to join the network and to send messages on it, and procedures to compute network configuration parameters. Thus, P2PS provides no more than the Message Handler and the Router.

Consequences

The benefits of the Peer pattern are:

- *Reusability*: all nodes share the same architecture, and are distinguished only by their runtime configurations, *i.e.* the Resource Provision Services they share, their roles on behalf of peergroups, Scheduler and Resource Monitor configurations, etc.
- *Efficient workload distribution*: resource discovery and provision is performed by dynamic groups of cooperating Peers.
- *Lack of single points of failure*: information about shared resources is evenly distributed and replicated.

The pattern has, unfortunately, some liabilities:

- *Complex application design*: the absence of centralized resource indexes requires efficient distributed search algorithms. Moreover, Resource Provision Services must be able to serve a request on their own, or by cooperating with other Peers.
- *Hard testing and debugging*: flow control oscillates between modules intra- and inter-Peers, preventing "single-stepping" through the runtime behavior of the network.

Related Patterns

The **Grid** [8] architectural pattern consists in using a middleware to manage distributed and possibly heterogeneous computational resources, in an efficient and transparent manner. In a typical Grid organization, participants have different roles: the user node (provided with a User Interface) submits jobs to a central coordinator (provided with Scheduler, Resource Monitor, Security Manager) which distributes tasks for execution on resource providers (each one with its Resource Provision Service). The Peer pattern can realize a Grid of nodes which are all provided with the same functionalities, *i.e.* job submission, resource discovery, resource provision, security.

The **Broker** [5] architectural pattern can be used to achieve better decoupling between clients and servers, in distributed software systems. The broker component is responsible for coordinating communication, such as forwarding requests, as well as transmitting results and exceptions. The broker is also able to locate the services required by a client, and to route the requests to the appropriate servers. CORBA [16] is the most famous implementation of the Broker pattern. In the Peer pattern, peer applications themselves realize a distributed broker.

The **Chain of Responsibility** [11] design pattern decouples the sender of a request from its receiver, by giving more than one object (disposed in a chain) a chance to handle the request. Requests can be issued without specifying the receiver explicitly (the set of objects that can handle a request should be specified dynamically). The Chain of Responsibility pattern can be used also if objects are distributed, maybe in conjunction with the following patterns.

The **Forwarder-Receiver** [5] design pattern provides transparent inter-process communication for distributed systems, introducing forwarders and receivers to decouple participants from the underlying communication mechanisms. In particular, forwarder components send messages across process boundaries, while receiver components are responsible for receiving messages. Both kinds of components provide a general interface that is an abstraction of a particular IPC mechanism, and includes functionalities for marshaling and delivery of messages. All services which need to operate on behalf of heterogenous Peer's instances, *e.g.* the Router, should use the Forwarder-Receiver approach.

The **Asynchronous Completion Token** [18] design pattern efficiently dispatches processing actions within a client, in response to the completion of asynchronous operations invoked by the client itself. For every asynchronous operation that a client invokes on a service, an asynchronous completion token (ACT) is created, uniquely identifying the actions and state necessary to process the operations completion. The ACT is passed along with the operation to the service. When the service replies to the client, its response must include the ACT that was sent originally. The client can use the ACT to identify the state needed to process the completion actions associated with the asynchronous operation. This solution is used by many event-driven modules of the Peer pattern, when their operations are invoked asynchronously.

Acknowledgments

This work has been supported by the "WEB-MINDS" FIRB project of the National Research Ministry.

References

[1] S. Agarwala, C. Poellabauer, J. Kong, K. Schwan, and M. Wolf. System-Level Resource Monitoring in High-Performance Computing Environments. *Journal of Grid Computing*, 1(3):273–289, September 2003.

[2] L. Onana Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N,k,f) A family of Low-Communication, Scalable and Fault-tolerant Infrastructures for P2P applications. In *The 3rd International workshop on Global and P2P Computing on Large Scale Distributed Systems*, May 2003.

[3] S. Androutsellis-Theotokis and D. Spinellis. A Survey on Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[4] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley & Sons, 1996.

[6] B. Carton and V. Mesaros. Improving the Scalability of Logarithmic-Degree DHT-based Peer-to-Peer Networks. In *Euro-Par 2004, Pisa, Italy*, September 2004.

[7] E. Christensen, F. Curbera, G. Meredith, and S. Weerawanara. Web Services Description Language (wsdl) 1.1. http://www.w3.org/TR/wsdl.

[8] R.Y. de Camargo, A. Goldchleger, M.R.F. Carneiro, and F. Kon. Grid: An Architectural Pattern. In *PLoP 2004*, September 2004.

[9] P. Dyson and B. Anderson. State Patterns. In *Pattern Languages of Program Design 3*, pages 125–142. Addison-Wesley Longman Publishing Co., Inc., 1997.

[10] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), March 2001.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[12] IETF. Public-key Infrastructure (X.509) Working Group homepage. http://www.ietf.org/html.charters/pkix-charter.html.

[13] Microsoft. Microsoft .NET Homepage. http://www.microsoft.com/net/.

[14] Sun MicroSystems. Java Remote Method Invocation (RMI). http://java.sun.com/products/jdk/rmi/.

[15] Z. Németh and V. Sunderam. Characterizing Grids: Attributes, Definitions and Formalisms. *Journal of Grid Computing*, 1(1):9–23, December 2003.

[16] Object Management Group. The Common Object Request Broker Architecture (CORBA/IIOP). Technical Report 3.0.2, Object Management Group, December 2002.

[17] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic Matching of Web Services Capabilities. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 333–347, 2002.

[18] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.

[19] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J.-C. Hugly, E. Pouyoul, and B. Yeager. Project JXTA 2.0 Super-Peer Virtual Network. *Project JXTA*, November 2003.

[20] Universität des Saarlandes, Swedish Institute of Computer Science, and Université Catholique de Louvain. Mozart Platform Homepage. http://www.mozart-oz.org.

[21] M. Volter, M. Kircher, and U. Zdun. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. Wiley, 2004.

[22] W3C. Web Services Architecture. W3C Working Group Note, February 2004.

[23] Web Services Resource Framework (WSRF) TC. The WS-Resource Framework. Technical report, OASIS, March 2004.

[24] R.J. Wieringa. *Design Methods for Reactive Systems*. Elsevier, 2003.

[25] S. Ye, F. Makedon, and J. Ford. Collaborative Automated Trust Negotiation in Peer-to-Peer Systems. In *Fourth International Conference on Peer-to-Peer Computing (P2P'04)*, August 2004.