

P2P Patterns

Results from the EuroPLoP 2002 Focus Group

Ekaterina Chtcherbina, ekaterina.chtcherbina@siemens.com

Markus Völter, voelter@acm.org, www.voelter.de

Version 0.4, December 14, 2002

Introduction

P2P systems are not a new concept. There have been p2p systems for a long time. However, due to the hype around Napster+Co, p2p systems are something the community talks about again.

Historically, many of the issues discussed in the context of p2p systems today have been discussed and inherent problems of distributed systems in general (for example, see [ST01]). With the advent of things like web_servers, EJB, .NET and recently, web_services, a more centralized approach has been taken to distributed systems - essentially client/server in many cases. The original issues with fault-tolerant, dynamic distributed systems now ship under the new name peer-to-peer systems. Here, Fault-Tolerant CORBA and DataParallel CORBA specify relevant topics.

There are some popular p2p systems such as Napster or SETI@home, however, it can be argued that these systems are not really p2p, technically speaking. Of course, from a user's point of view, they exchange files with peers, other users, but technically many of these systems are more or less client/server with the "shortcut" of exchanging large-volume stuff without bothering the central server.

So, what then characterizes a p2p system technically? And what are some of the issues that have to be solved to construct "real" p2p systems? The following paragraphs try to shed some light on these questions, based on the results of the focus group on P2P patterns at EuroPLoP 2002 in Irsee, Germany.

Important Characteristics of P2P Systems

The basic concept that distinguished p2p systems from traditional client/server kind of systems is the fact that all nodes in a distributed system are *considered equally important*. Each node is able to provide some services to other nodes and there is no specific "most important" node. This does not mean

that every node has to provide the exact same set of services. There may very well be some nodes that serve service A while other nodes provide service B. The important point is, that all nodes are technically able to provide services for use by other nodes – *client and server are merely roles with respect to one specific service relationship*.

As a counterexample, consider a typical client/server scenario: client (such as web browsers or fat clients) are not technically able to provide services to other clients or the server(s). Servers, on the other side, are not “active” in the sense that they drive the system, they just provide services (such as a database, e.g.) to clients.¹

This has several consequences. First of all, all the nodes have to speak a *common language* and use common semantics for their protocols. This may seem trivial and obvious, but it is an important issue. You cannot build a p2p system when different nodes use different protocols or even different versions of the same protocol. *Version management* in a distributed system becomes an issue.

To simplify matters, protocols and the transported data should be *self-descriptive*: this means that a message also contains meta data that describes the meaning of the message, in whatever format seems useful (note that the nodes have to agree on the format and the version of the meta data!). XML-based descriptions can be used to good advantage here. Again consider a counterexample: Pure, plain CORBA which uses IIOP as the transport protocol does not contain metadata describing the interfaces and operations, for performance reasons. A message sent from a client to a server can only be interpreted correctly if both parties use stubs and skeletons generated from the exact same IDL file.

As a consequence of the fact that any node can provide and consume services, a sophisticated *service relationship management* has to be put in place. A node must be able to find, and then use, the services it requires. Typically, the services provided and consumed by a specific node are not known a priori – the service relationship management must be *dynamic*. Nodes joining and leaving the system, or nodes that change their service portfolio need to be coped with. This also includes the problem of distributed garbage collection: how can a node determine that a specific service is not required any more and can be safely removed? Concepts such as Leasing provide a solution to this problem in many cases.

Because all nodes are considered equally important, no failing node is allowed to bring down the whole system. Of course, clients using services provided by a crashed node may notice a problem – but the system as whole must not break.

¹ In many real-life systems this distinction is blurred, of course. For example, a web browser can be seen as a server providing an “HTML-rendering service”. However, this is a very specific and generic service that is not usually treated as a “service” in an application architecture.

To allow for that, the p2p system must *dynamically readjust* its own configuration. Note that this must be true for any failing node, there must be no single points of failure. So also the configuration information must be decentralized.

Which brings us to the next issue: the *state* of the system is typically also *distributed* over (some of) the nodes in a peer-to-peer system, raising issues regarding replication and synchronization.

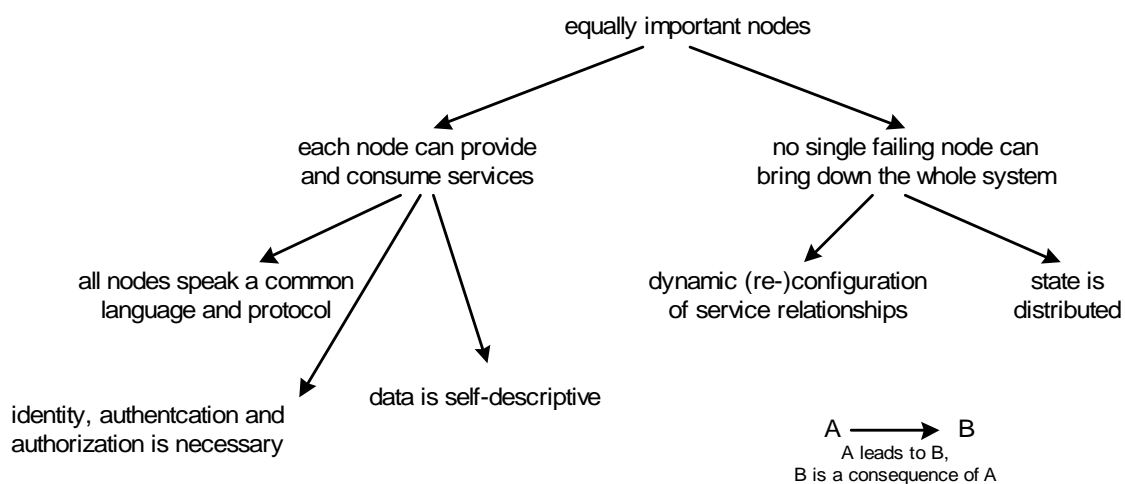
Last but not least, *security* is an issued: How can it be determined who is allowed to provide or consume services? How are specific peers or their users identified? Managing trust relationships in a p2p system is one of the most critical issues here.

So, summing up, a P2P system is characterized by the following items:

1. It consists of conceptually equally important nodes
2. Each node can provide and consume services
3. These service relationships must be dynamic and readjustable
4. No single failing node (or service) can bring down the whole system

Note that the items 2 through 4 could be considered consequences from the first item. And the other issues mentioned above (self-descriptive data, security, replication of state and the common language) could be considered consequences also.

So, the only real requirement is: A P2P system is a collection of equally important nodes collaborating to do something useful, with ability to dynamically reestablish relationships.



The next section outlines the technical challenges involved in implementing the characteristics above. Wherever possible, we reference already existing work and patterns to hint at solutions.

Optional issues / challenges

Here we show the problems involved with the issues mentioned in the above section and of those below. Provide pointers/solutions/patterns to help solve them (I have a great book (by Tanenbaum) on distributed systems that describes many strategies for synchronization, data consistency, elections, etc.).

Business models, payment, roles and legal issues

In order to bring P2P systems into a real life we need suitable business models that would allow providers of P2P solutions to benefit. P2P provides mechanism to build extremely flexible systems - no pre-determined system players, no pre-determined users. Therefore, it is also extremely difficult to ensure payment mechanisms in this dynamic environment. The following issues can be identified.

Because every peer can provide and consume services, each peer must be able to pay for services, but also to receive payments for their provided services. Every peer needs to be capable of charging (calculation of the price), billing (sending a bill) and payment (paying according to the bill). An efficient, secure and lightweight payment infrastructure is required that can be deployed on each peer. This has serious consequences for the security mechanisms on these peers. In today's Internet system, payment is often delegated to central payment providers with whom the service provider has special contracts to avoid the problems of payment transactions on each web site. In peer-to-peer systems this approach is not so easily possible.

Peer-to-peer systems are not built on a predefined, centrally administered infrastructure. Every peer in the system is part of the infrastructure. Consider a peer A that uses services from peer B. Using B's services might include the intermediate services or resources of many other peers, primarily of technical reasons such as routing, lookup, etc. Solutions are needed to "pay" these intermediate peers for their services. Note that the amounts for such services are probably rather low. This means that we need a micro-payment system that allows to handle many small payment transactions. Today, this is not available on large scale.

There are other relationships with security issues. If peer A wants to use a service from peer B and has to pay for it, then A wants to be rather sure that it is actually using the real B's service - authentication is necessary (see below). This is also true for the subsequent payment transaction.

Ownership of shared data in P2P systems is another challenging issue. If peer A buys data from peer-provider B, but then supplies this data to peer C then in certain cases peer B would like to get a certain part of C's payment to B. Or, B does not allow A to publish its own data to other people - C in the example. How do you ensure this (this probably sounds familiar: this is part of the current debate with the music industry).

We need to define roles in P2P systems, where a seller can act as a buyer at the same time. On the other hand, in case of end-user acting as a provider, QoS and trust becomes an issue for a buyer of the service. Ownership rights and license agreements become more complicated and require mechanisms for distributed non-deterministic contract management.

Security Concerns

Security is an issues that is non-trivial in all kinds of systems. Security comprises things like

- *Authentication*: making sure a user of the system is really the one he claims to be. In peer-to-peer systems service and resource consumers might require proof of information about the provider, otherwise authentication cannot be considered successful. Therefore, distributed trust establishment mechanisms are needed.
- *Authorization*: deciding who is allowed to access what. In centralized systems the user rights are pre-defined and therefore the decision to allow access for the certain user is taken based on these pre-defined rights. In P2P systems the requestor is not known a priori, that leads to a complex decision making process. Different approaches are possible in decision-making: based on recommendation (I know peer C, peer C recommends peer D, therefore I can give an access for peer D), or base on the experience (I have already given access to the peer with the similar characteristics based on recommendation, therefore I can give access to this peer as well).
- *Encryption*: making sure data cannot be read by non-authorized parties, making sure it was not changed "on the wire" without this being recognized, proofing from whom the data came, for example with cryptographic signatures, or making sure that actions that have been executed cannot be claimed never to have happened (non-repudiation).

Note that security is a really important aspect in P2P systems because the user community is potentially open. Whereas in centralized systems you typically have a rather small, typically well-known set of users, peer-to-peer systems are conceptually extendable. There is often no way to "keep people out" easily. Thus, the system must be especially hardened against insider-attacks, because people can very easily become insiders.

In traditional systems, security functionality is often implemented as a central, small, reliable so-called *trusted computing base (TCB)*. A TCB is typically a system that has been thoroughly analyzed and tested as is therefore believed to be tamper-proof. In distributed systems, and especially in peer-to-peer systems there might not be central instance to check security concerns. As a consequence, the TCB of the system is itself distributed and becomes much harder to analyze and test – it is therefore much less trustworthy.

Authorization is also harder: consider, for example, the situation where you don't have a central user database: In such a case, users and permissions are often based on a trust-system. Simply put this means that if there is a (unknown) user A which a known, trusted user B trusts, then I, user C, will also trust A. Distributed authorization can be handled in a similar fashion, or, alternatively we can use a decentralized, distributed user database. Then the problem is reduced to that of keeping data in a distributed repository – although with the additional constraint that this is actually very important data. Then the database with user rights is distributed and there is no problem, when one of the peers in the network fails. But still there is problem, when the peer that is not yet known by other peers of the network has to be authorized because specific criteria must exist for such cases.

Encryption in peer to peer systems is not harder, in principle, than in any other use case once the parties have agreed on protocols and algorithms and on the way how initial keys can be distributed. This is because encryption is always considered to be something that is happening between potentially untrustworthy parties – peer-to-peer is no additional constraint in most cases.

State and data management

Peer to Peer systems are characterized by the fact that a single failing peer must not bring down the system as a whole. Of course, specific services (those that had lived on the dying peer) might not be available anymore, but the system still fulfills a useful purpose. In many systems this requires facilities for some kind of distributed data management. As a consequence, we have to look at the following issues: replication, caching, consistency and synchronization, and finding the nearest copy.

Let's first introduce two concepts: Hard and soft state. Hard state is data that cannot easily be reassembled. It is crucial that such data not be lost. Soft state, on the other hand, is state that can be rebuilt from other (typically hard) state. Caches, by definition store soft state only.

Let's describe the ideal situation first: Ideally, every data item is stored on every node, and it is always up-to-date. Obviously, this is not a realistic option for all but the simplest systems. So the first issue is to decide which peer will actually store what data. A reasonable strategy would be that a client stores

only data related to services that it is really interested in because this store would then automatically also serve as a cache. An alternative solution is that each client provides a certain amount of storage space where the underlying peer-to-peer middleware can store arbitrary data. Then, of course, data must be represented in a way that allows peers to store the data of any service - XML and meta-data come into mind.

Of course, once data is stored in more than one location, we need a way to determine how to keep the multiple copies up-to-date, synchronized and consistent. This involves two main issues: how fast are updates propagated(?), and how do we resolve conflicts. It is not realistic, of course, to propagate (?) every change to every peer at once, this would incur too much performance overhead. So we have to define policies for each data item (or data type) when it is replicated and who initiates the replication (push or pull). If data is pushed to peers, but not instantaneously, then peers have to cope with the situation that data might be out of date. Also, the replication strategy should take into account subnets that are likely to work separately from time to time, for example because some devices are temporarily offline or because a WAN connection is not available all the time.

Regarding conflict resolution, there are two principally different ways how to handle this issue: The first way is to avoid conflicts in the first place by making sure there is a primary copy of each data item. Changes are always applied to the primary and then forwarded to the other replica. Changes to other replica are not allowed. Of course, in a peer to peer system we never know which peer (and thus, data copies) are currently online and we must redetermine the primary copy of a data item on a regular basis (see election algorithms below).

A primary-based algorithm requires that all peers who want to change a data item need access to the *one and only* primary copy. This might not be realistic in peer to peer systems where peer groups, group memberships and network configurations might change on a regular basis. Whenever such a primary-based approach is not realistic, we could also allow each peer to change the data in its local (or nearest) replica. However, then we have to find an algorithm how conflicts are resolved, because several peers might apply different changes to a specific data item on different replicas. In practice, such conflicts cannot be resolved automatically, application specific code is necessary. However, a peer to peer middleware should provide support for replication and conflict detection, subsequently calling an application level hook to resolve the problem.

Another way to resolve conflicts (or, really, to prevent them) is to use pessimistic locking and transactions. Distributed 2PC and 3PC protocols can be used - of course the peers have to agree on a transaction coordinator for this purpose. However, techniques based on pessimistic locking (ACID transactions fall into this category) are not very well suited for peer to peer systems.

For reasonably large systems, replication of data cannot be extended to every peer. Only specific peers will store certain data items - we have discussed ways to decide which data to replicate where above. However, as a consequence, a peer who wants to access a data item has to determine which replica to use. This decision should be based on performance considerations: A peer should use the data item that it can access most efficiently (read: that's "nearest" in network/bandwidth terms). It is not always easy to find out which replica is actually "nearest". Basically, this boils down to a lookup/discovery problem discussed below.

The above discussions focused on hard state and its replication for the purposes of failover and QoS. Independent of all these discussion can clients keep their own copy of a data item for caching purposes. Read-only caches are uncritical regarding consistency. Read-Write caches are more complicated because they behave like a changed replica (see above). Write-through caches boil down to a primary-based approach as discussed above.

Up to now we have omitted one very important issue: how we get rid of a data item (a cache entry or a replica) when we don't need it anymore? In this context, we can regard data as being a resource (just as a provided service, etc.); see the discussion of lifecycle and distributed garbage collection in the context of peer to peer systems below.

Lifecycle Management and garbage collection

It is typically relatively easy in distributed systems to set up a resource, and allow clients to connect to it. The harder part is to make sure that unused resources are eventually removed to make sure the system is not filled up with garbage. The discussion that follows applied equally well to different kinds of resources, be they provided services, data (replica), group definitions, or communication sessions. The problem is generally known as distributed garbage collection. In traditional systems, distributed GC is difficult because of the fact that a part of the system might fail unexpectedly. This might of course also happen in peer to peer systems, but there we have the additional problem that peers become offline as part of their regular operation. As a consequence, an unavailable peer is not a consequence of a (more or less seldom happening) failure, but part of the ordinary operation. Therefore, more efficient algorithms are required.

There are many different algorithms used to implement distributed GC. The most important ones are based on the concept of clients expressing their continued interest in a resource - otherwise the client is considered as not being interested in the resource any more. For this reason, a resource-provider issues leases to clients. A lease is a time-limited permission for the client to use the resource. Once the lease has expired (e.g. the client did not request extension of the lease) the client has no right to access the resource any more. A resource provider tracks the clients and their remaining lease time. Once the lease time

reaches zero, the provider removes the particular client. When no clients remain, the resource can safely be removed.

There are different flavors of implementing leases:

- The traditional heartbeat is a very primitive form of leases.
- The resource provider might be allowed to reject a request to extend the lease (useful for services which the provider wants to shut down in an orderly fashion) or only allow a shorter period than the one requested by the client.
- When a lease time reaches zero, the provider could get back to the client and ask him if he's really sure that he's not interested in the resource anymore.

In larger systems it is not always simple to come up with a reasonable lease time: small time periods allow for faster reclaiming of resources but increase network traffic. In such cases, more advanced, hierarchical management of leases might be required. Also, mechanisms have to be put in place to handle the case where a peer provides a resource he leased himself to another client. This can either be disallowed altogether, or more elaborate algorithms are needed to handle this case.

Dynamic Service Relationships

Dynamic Service Relationships become an important issue in p2p systems due to the fact that those systems are non-deterministic, dynamic and are self-organizing based on the immediately available resources. A p2p system is typically loosely coupled, moreover it is capable of adapting to changes in the system structure and its environment: number of peers, their roles, and infrastructure.

In order to build a loosely coupled system that is capable of dynamic re-configuration, several mechanisms should be in place:

- *Discovery*: there must be a distributed search mechanism that allows for finding services and service providers based on certain criteria. This is the main pre-condition for building a self-organized system that will be able to react to changes in the system structure. Having a look-up service (such as the CORBA federated trading service) solves part of the problem of finding required resources but leads to hybrid system, where all resources have to be registered before being used by other services or peers. Nevertheless, this solution works well in wide area networks. The challenge in this solution is to find the right number of look-up services that should be available in the system. The load resulting from the number of lookups on a specific service instance and the overhead through replicating the publishing information (lookup services can also be seen as a replication problem – see above). Another challenge is how

to decide which peer will run a look-up service in a fully distributed environment. Again we need a decision-making system or voting (see coordination issues below). Running a look-up service requires additional resources such as power and memory from the peer, therefore cannot be always requested from the peer on a free of charge base (see business models).

A fully distributed discovery mechanism also raises other questions such as the achievable efficiency of searching. Also, it is hard to decide when we should stop searching - we do not know whether the resource that we are looking for is not available or the way of searching that we have taken is just not the optimal one: there are different approaches for discovery that can be applied based on the topology of the network. The two most famous of them are caching in hash-tables - a method that is applied in content-based networks, - and publish/discovery mechanisms. The first one tries to optimize search performance by using distributed caching, and the second by trying to find a right balance between advertising (publishing) and search (discovery) messages, otherwise too many resources are consumed. Although these methods are not newly invented and have been applied earlier, the discovery mechanism is not that efficient as it should be to be used in real-life (large, widely distributed) systems.

- *Naming /Addressing*: in order to identify a resource (peer or service) a unique identification mechanism or naming concept needs to be introduced into a p2p system. How to address a peer in the global network? Addresses that are normally used to access the node in the network (such as IP-address in the TCP/IP network) do not help a lot - the p2p system is heterogeneous, therefore different addressing protocols can be theoretically used within one p2p network. Again, CORBA provides some solutions here: CORBA's IORs provide the capability to store information about multiple ways to access an object and mobile CORBA addresses mobility issues such as changing IP addresses in case of migration from one network domain to another.
- *Self-description* of the data: systems with a fixed infrastructure as well as fixed subsystems grouping are based on pre-defined relationships. P2P systems being loosely coupled systems must provide mechanisms for data self-description in order to build relationships between services. The challenge of self-description is in definition of the level of details of the meta-language. Indeed, can we call a system that for a certain request will accept only the certain response and will ignore all responses that are not pre-defined a priori? There is a challenge of interpretation of the resource description written in a language not necessarily known by the resource requesting it. The problem is how to introduce new data that is not known before. The system should be able to understand that this new data is "some kind of something I already know". There are ways to

described such things using ontologies. However, then the problem becomes how to describe the ontology, in case *it* changes. We'll need meta-ontologies here...

- *Negotiation/Trading*: the consumer of the p2p resource is end-user or another p2p resource, we need a mechanism that will allow for finding the optimal available resource to be consumed based on some criteria that is given by the consumer. The task can be solved by introduction of suitable protocols, but the negotiation in the distributed environment needs again a complex decision-making mechanism on both consumer and provider sides.

Note that this complete dynamism does not make sense for all kinds of services. Consider a service that is used by other services to (persistently) store their data. The purpose of this service cannot be realized any more if we assume that this service is online or offline whenever it wants. There is simply no point in storing data somewhere, when this "somewhere" will eventually not be reachable any more. Of course, we have to protect against failure and for example replicate the stored data. But we have to assume that the service (or the group of replicated instances) is available all the time. Note that this is not a constraint on the whole system, but only on some services, whose whole purpose is being available.

Fault Tolerance

The term *fault tolerance* means that a system can provide its services even in the presence of faults² that are caused either by internal system errors or occur due to some influence of its environment. P2P systems are used in situations when a system has to function properly without any kind of centralized monitoring or management facility. Therefore, automatic *self-recovery* from failures without seriously affecting overall performance becomes extremely important for P2P systems. Sometimes it is not possible to recover from a failure, however. It is then necessary that the system be capable of adequately providing the services in the presence of such partial failure. In case of a failure a P2P system must be capable of providing continuous service while necessary repairs are being made.

In order to build such a fault tolerant P2P system, several issues have to be addressed:

- *Reliable Processes*: one possibility to achieve fault tolerance in P2P systems is the to make sure a single failing process does not lead to service unavailability. The easiest way to implement this requirement is to have several (more or less identical) copies of a process. Identical

² We do not distinguish between bugs, failures, etc. Instead, we use the term *fault* to denote any case of "malfunctioning".

processes can be organized into groups that are responsible for the provision of certain services. In such a scenario, a request is sent not to one single process, instead it is multicast to all processes within the respective group. Any member of the group that is capable of providing the service can take on the responsibility.³ This approach works only if reliable multi-casting of messages within a group is available, and if there is some kind of synchronization mechanism among the members of a group is available. If one of processes crashes at one peer another process takes over and provides a requested service. The other process can be on the same peer or (typically) on a different one. The allocation of processes to groups can be also be done dynamically, then some kind of group membership and group managing is required – again, without a centralized manager. Introduction of groups allows us to deal with a set of processes as a single abstraction, leading to a certain transparency for clients on the one hand while still providing a level of reliability of services on the other hand.

- *Multi-hop routing with recovery:* While the previous bullet point focused on providing resilience for process failures, this section considers partial failure of networks. Multi-hop message routing in distributed environments is a hot topic due to the challenges of routing in mobile networks that stem from changing topologies and device power limitations. In general, multi-hop routing means that the sender of a message does not directly send the message to the receiver, instead the message is forwarded by one or more intermediate routers. Algorithms are required that allow for efficient multi-hop routing considering the mobility of devices and therefore changing network topology, power consumption of the devices as well as (temporary) failure of network nodes or routers. As of today, multi-hop routing is not supported natively on the transport layer, thus today the network must be built taking into account the limitations of the underlying hardware, e.g. 100 m for the wireless LAN. If a network node moves out of the range, there is no possibility to send a message to it, although a rule of transitivity could be applied in practice – if node A sees node B and node B sees node C, this implies that node A sees node C. But for that kind of routing, smart algorithms are required. The hard problem here is how to send the message avoiding loops and infinite routes? In choosing a node as the next intermediary for a message, a certain optimization algorithm must be applied in order to find the “best” possible route. For example, in an ad-hoc network where peers have limited power resources a peer could send the message to the node with a high *potential* (in terms of

³ Note that sometimes not all processes are identical at any time; one might be actively serving requests while the others merely listen what’s going on. Election techniques as discussed further below can become necessary.

power) that is chosen from all available nodes. . At the same time the route must be cached at each node in order to discover cycles and to be able to send an acknowledgement for the message back to the original sender. The problem, however, is that at some point, no node with a high potential might be available anymore, leaving the message stuck in a dead end, not reaching its goal. These examples illustrate that multi-hop routing cannot simply follow the techniques that are employed in wire-line networks, but should be based on a multi-route approach.

Fault tolerance for routing mechanisms in P2P networks is actually not as important as the ability to self-recover of the system. Imagine that communication fails due to the failure of a routing service (lost/removed intermediate nodes, failure of routing calculation etc). In that case recovery of the situation (re-calculation of the route) becomes extremely important to keep message delivery alive: the system must be able to find a new route to the destination. There are two ways to do this: backward recovery and forward recovery. Backward recovery tries to bring the system back into a previous correct state, requiring recording of such correct states at some kind of a centralized storage. In case of forward recovery the system attempts to find a new correct state that will enable further successful activity of the whole system instead of coming back to a previous “successful” state. This is the same task as finding a new route for a message in the context of changing network topology.

Redundancy as you have seen is a key technique to achieve fault tolerance of the system. Although redundancy leads to the “un-efficient” use of resources (storage, bandwidth etc) it is the only way to achieve a certain level of reliability of the overall system. In P2P systems, especially in mobile, embedded devices, both the extra use of resources and the desired level of fault tolerance need to be taken into account. Only by balancing these two conflicting forces appropriately for a specific scenario can the desired behavior of the system be achieved.

Mobility

Taking the dynamic aspect of P2P networking into consideration, the devices in a P2P systems are often mobile. That includes physical mobility as well as logical changes to the overall application structure. On the application level there is a need to support changing network technologies (LAN, WLAN, Bluetooth, GPRS) and addressing in a mobile environment.

There are several aspects that need to be discussed:

- *Mobility of peers and services*: mobility means that participants of the network are changing their location while the system is operational.

When talking about mobility, scenarios of cars rushing through the streets of a city or people walking around in public places such as shopping centers or stadiums are typically used for illustration. But mobility can also mean that a device is in one place for several hours and is then moved to another place to reside there for another couple of hours. Each time a network node moves (or is moved), the entire topology of the network changes. Two nodes that were just close together and could easily interact with each other might be far apart in the next moment requiring some dedicated “routers” in between in order to establish any kind of communication, and experiencing increased latency. Or it might happen that a node moves out of the scope of the network and loses the connection completely, while other nodes come into reach and want to participate. Most P2P systems need to be able to work under these circumstances. Also mobility usually implies that devices just have limited electrical power and limited network bandwidth, requiring efficient use of these two resources.

Mobility of services brings additional constraints to the design of a P2P system when we take QoS and availability/reliability issues into account. The requestor of the service expects to get a certain level of service quality and also expects to have the service available when needed. This is a challenge in mobile distributed environments when services are not only distributed but also changing their locations. One approach of tackling this problem is to implement “session hand-over”, providing mechanisms for dynamic re-delegation of the service provider role to other peers.

The mobility aspect requires a “smart” support for wireless networking technologies. This means that an application must analyze the problems on the transport level and provide smart solutions for them. One of the problems in the wireless standard IEEE802.11 is that TCP performance goes down due to bandwidth limitations. Ad-hoc mode of the same standard has several problems with multicasting – messages can be lost due to the loss of connectivity and this leads to the performance problems due to the re-sending of packages. In order for the system to work efficiently with those wireless technologies, the system must be aware of the problems and react properly. If we look at advertisement of resources in an ad-hoc environment, we need to find a proper frequency of advertisement messages that should be multicasted in the network in order to successfully propagate a resource on one hand but not to reduce performance of the network due to the lost packets on the other hand. This is an issue that can be solved only if p2p system is aware of physical connectivity and can adapt systems behavior to the environmental change. *Address resolution:* we have discussed mobile aspects of P2P systems. This leads to a discussion of addresses and name spaces in a distributed environment. In order to access nodes in P2P

networks we need an identifier that is independent of the network-topology, the concrete transport used (as there may be many transports on the route) and of the geographical location. An identifier always refers to one and only one entity, therefore provides a unique “access point” for a peer in a P2P system. The last but no less important issue that has to be solved in distributed, mobile P2P environments is mapping of identifiers to physical addresses taking into account that we are dealing with a large-scale, possibly worldwide-distributed system. It is useful to distinguish global, administrative and managerial layers in the P2P network. Nodes in the global route are characterized by high stability – they typically represent organizations, don’t move and are very reliable. Nodes in the administrative layer belong to the same group or administrative unit. Nodes in managerial layer typically change regularly. Considering this 3-layer model, we can conclude that the aspect of mobility mostly affects the managerial layer. Here it becomes important to choose names that can be expected not to change during the lifetime of the entity they represent. An even better solution is to separate naming from locating by introduction of identifiers. In that case we need a service that will take an identifier as an input and return a current address/location/route of the identified entity.

- *Geo-location*: In its extreme form, P2P systems can have global scope, where nodes from all over the world can communicate with each other. How important is the (current) physical location of a peer in P2P networks and when should this issue be taken into consideration? One important case is using the location-information on the application level, for providing location-based services such as printing, shopping, city-info etc. Management of groups and service lookups can also be improved by location-awareness. Another issue of geo-location concerns synchronization of services and communication. As an example of required synchronization, take a look at scheduling access to a shared resource: requestors receive a time-limited access grant (or lease) for the resource. In order to fulfill the specification of the system and to get the job done, synchronization in terms of time is required. This is much more difficult than in a centralized system as there is no notion of a globally shared clock/state. This leads to the requirement of distributed mechanisms for clock synchronization, or at least to be able to decide which events have occurred before/after other events.

Coordination

Coordination requirements come in many different flavors in P2P systems. Examples are:

- The distribution of load among a set of (similar) peers. When a request for a service needs to be fulfilled, there needs to be a way to determine

which of possibly many service providers will serve the request. Such a situation occurs primarily when several providers can service a request for load-balancing purposes.

- Failover scenarios typically involve the distinction between *active* peers and *passive* (or *standby*) nodes. Active nodes typically serve client requests, whereas passive nodes usually listen what's going on in the system, or update their internal state from the active node. Assuming nodes are stateful, which means that only one of a group of providers can be active at a time, somebody has to decide who is active, and who is passive.
- Distributed 2-PC transactions are another example: the transactions have to be coordinated (begins, rollbacks and commits) by some transaction coordinator.

Now, the items listed above are not (yet) specific for P2P systems. What makes them interesting, and specific to P2P systems becomes obvious when we reconsider some of the basic assumptions underlying P2P systems:

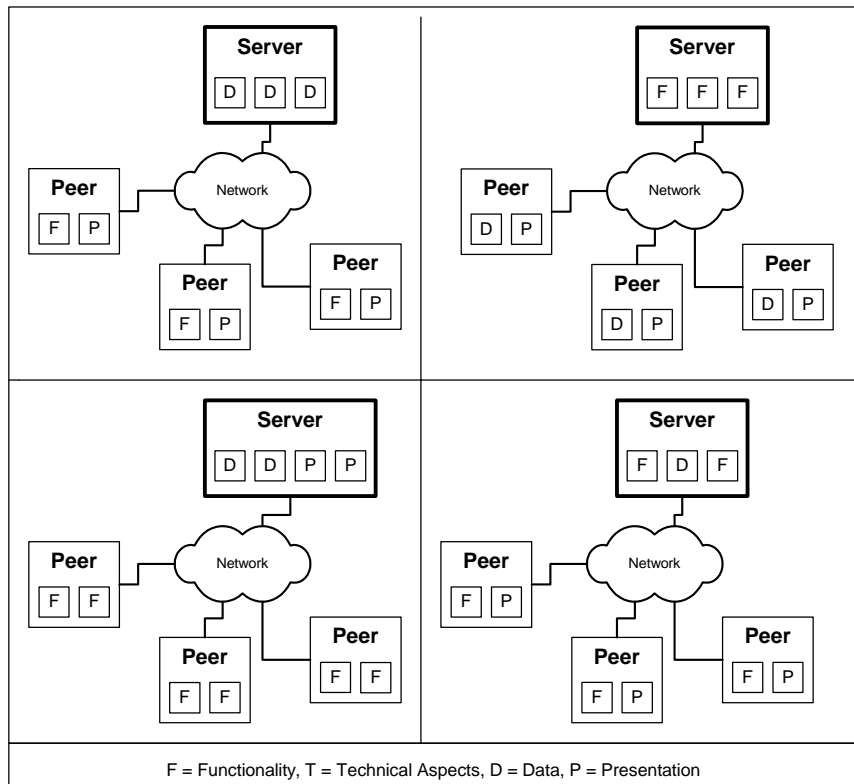
P2P systems are designed not to have a single point of failure – a certain level of service quality must be provided under all (but catastrophic) scenarios. Also, P2P systems are characterized by the fact, that *it consists of conceptually equally important nodes*. This means that there are no central servers and managers on which the system depends. However, all the above mentioned coordination issues depend on a central entity...

This conflict can be resolved, however. Our definition of P2P systems does not say that all peers must all provide the same set of services/functions at any time, it only says, that the different nodes in a system have to be *capable* of providing the same services. Specifically, this means that some peers can play a special role in the system for a specific period of time. The important factor is that whenever this “special” peer fails, the system can come up with a decision on who will subsequently play the special role – without requiring a central coordinator for this. What this comes down to is distributed election algorithms. The system must be able to come up with and enforce decisions about who plays special roles without requiring a predefined arbiter.

The literature describes several distributed election algorithms, each with particular tradeoffs regarding the time required to come to a decision and the overhead (regarding performance, network messages, etc.) involved in it. As usual, these tradeoffs have to be resolved in a manner specific to the problem at hand. What makes this interesting in the face of P2P systems is the potentially large number of participants and their limited resources (bandwidth, etc), combined with the dynamics of the changing topologies.

P2P, Client/Server and Hybrids

A final note on the “equality” of peers in a system. We have said that every peer in the system must be able to provide the same set of services. This is true for pure P2P systems. However, there are definitely some useful hybrid systems, that should still be considered P2P, in principle.



For example, bulk data can be stored on central servers, central servers can provide some central indexing of distributed data, they can serve as a task coordinator and they can be used to actually do expensive computations.

To keep this system P2P in its basic setup, we can define two domains: a server domain and a client domain. Each has specific roles according to their intrinsic capabilities (big server machines can do other kinds of things as small PDAs). For each of the domains, the basic P2P principles apply:

- clients can join and leave the client domain, they can move around, provide store amount of data, be involved in distributed transactions, join and leave peer groups, etc.
- the server domain is also built as a P2P system: all servers can provide all (server) services, they coordinate themselves whenever one goes down, they can be (re-)located all over the world, take part in distributed transactions, etc.

However, the clear definition (and separation) of the overall roles of clients and servers, and the way they (or their services) collaborate is more or less static. This gives us the best of both worlds: on the intra-domain level we go P2P, whereas the inter-domain relationship is classical client server.

Webservices

Typically, when discussing about web services, the discussion focuses on technologies such as SOAP and WSDL that (will) allow a standards-based client/server communication among heterogeneous platforms. Another, more interesting aspect, though, is the concept of UDDI which serves as a trader that helps to construct service relationships among webservice clients and servers. The vision of webservices is that providers (typically businesses) publish webservices to UDDI registries which clients can dynamically discover and use. UDDI aims to fully describe a webservice, allowing autonomous clients to use it sensibly. As such, Webservices constitute a form of P2P systems. It is interesting to look at the webservices topic from this perspective rather than looking at the used technologies for remoting.

Summary (specific to article)

In this article, we have tried to define what P2P systems are and we have outlined some of the challenges faced when implementing such systems. Many, though definitely not all of the issues have been solved in the computer science and software engineering communities, some only theoretically, others also with practical implementations. What is missing is the integration of all these things in a framework that is useable for widespread use in P2P systems – also on small devices on networks providing only limited bandwidth. Projects with this goal are underway, e.g. JXTA, but they still have a long way to go.

Summary (specific for report)

As already mentioned in the the introduction, we have not come to the point where we identified concrete patterns in the P2P world. This is a chance for future work, and we hereby invite every interested individual to help in this effort. However, we think that many (though not all) of the patterns have already been documented in the context of general distributed systems and coming up with a P2P pattern language should largely be a task of organizing adapting available material.

Acknowledgements

Thanks to all participants of the EuroPLoP 2002 focus group on P2P patterns, and specifically to Andery Nechypurenko, who reviewed this paper after the workshop.