



IP-FP6-015964

AEOLUS

Algorithmic Principles for Building Efficient Overlay Computers

Deliverable D3.2.2.

Microbenchmarking software package: Design report

Responsible Partner: Università degli Studi di Padova (I)
Report Preparation Date: September 2006

Contract Start Date: 01/09/05 Duration: 48 months
Project Co-ordinator: University of Patras (EL)

Contents

1	Introduction	1
2	Previous work	2
2.1	The microbenchmarking approach	3
2.2	Measurement techniques for Cluster-based parallel machines and the Grid .	4
2.3	Measurement techniques for the Internet	7
2.4	Topology awareness techniques for P2P Systems	12
3	Requirements for the Overlay Computer	18
4	Design and implementation of initial tests	21
5	Results	23
A	JXTA overview	30

1 Introduction

The impressive amount of computing and storage resources potentially available through the Internet has stimulated several research and business projects aimed at making it profitable for vast scale applications, the first and most popular being Peer-to-Peer (P2P) file-sharing applications. The opportunity to exploit idle cycles of connected computers also led to the development of several world-distributed applications, such as SETI@Home [45], GIMPS [16], and others. While these applications are essentially based upon an asymmetric producer-consumer paradigm, they constitute a witness of the potential offered by wide-area network distributed computing also for other, more general, parallel applications. The parallel computing platform that would actually deploy the computing and storage resources needed for this purpose, and whose network topology is embedded within the Internet is called an *Overlay Computer* (OC) and its basic principles, design choices and modes of operation are at the core of the AEOLUS project.

To be successful an OC must provide higher performance than the individual computing equipment available to the user (in the extreme case, access to an OC should enable a user to accomplish tasks otherwise impossible on his/her own platform(s)). To this purpose, it is necessary to provide the user with tools for estimating the effectiveness of design choices on such a distributed computer. The tools must rely upon performance metrics aimed at providing a guestimate of the key factors impacting performance of a distributed application, such as interconnection *latency* and *bandwidth*, and available *computing power*.

Measuring performance is a manyfold task basically used to collect parameters for the purposes of service evaluation, tuning of cost models, etc. The purpose of this report is to investigate the use of *microbenchmarking techniques* [41] to measure performance characteristics of an OC. These can be employed to:

- Improve performance of the *overlay topology* (i.e, the embedding of the network of peers onto the underlying physical network);
- Design performance-driven primitives for spawning processes over the OC and efficiently running parallel applications;
- Evaluate whether an application is well suited for running on the OC with a certain performance guarantee;

The above mentioned points have different (more or less stringent) requirements on the overhead and the accuracy required of the measurements and on the amount of computing resources that can be devoted to the microbenchmarking activity. For instance, the tests aimed at optimizing the spawning activity should return useful indications for choosing the peers where the individual processes of the parallel application should be allocated, in a sensibly shorter time than the tests aimed at globally optimizing the virtual topology

connecting the peers designated to run the application, since the last activity can afford more precise and longer measurements.

Optimizing the overlay topology is a fundamental task to increase the overall computing capabilities of the OC. Many research results show that substantial improvements can be attained when exploiting locality or when organizing the overlay topology based on the capacity of the peers as discussed in Section 2. The ability to adapt to the characteristics of the underlying network to improve performance is often referred in literature as *topology-awareness*. To be topology-aware, the OC should to be able to place a joining peer onto the overlay network exploiting both computing and communication capabilities of the peer itself. Therefore, the OC needs primitives to probe the performance of peers and their interconnections.

To be able to efficiently run parallel applications, *performance-conscious* spawning of the processes of a given application is a crucial feature of the OC. The spawning process should take into account parameters of the application (such as, for instance, the computational load, the degree of parallelism, the degree of redundancy to enforce fault-tolerance, the computation to communication ratio, etc.). For a good fit of the application over the network, the OC needs to match the performance requirements of the application with the parameters of the overlay topology, and thus these latter parameters need to be measured.

This report has two main objectives. The first is to provide an overview on performance measurements techniques for related parallel or distributed platforms such as cluster computers, the Grid, the Internet, and P2P systems, also discussing the role of performance consciousness in improving the performance of the various systems. The second objective is to provide a very preliminary set of experimental results to serve as *proof-of-concept* for the development of a more extensive microbenchmarking tool for the OC. The proposed tests aim at identifying both measurement techniques and key performance factors that any such tool will need to target in order to be effective for the OC.

The report is structured as follows: Section 2 presents a overview of performance measurement tools for cluster computers, the Grid, the Internet, and P2P platforms. The role of topology-awareness to improve performance of P2P systems is also discussed. Section 3 discusses the requirements of a suite of tests for an OC while Section 4 presents the tests implemented as preliminary effort towards a more complete microbenchmarking tool. Finally, Section 4 shows the results of performance measurements executed on a local area network running JXTA. The presented results highlight some of the performance issues of such a P2P infrastructure and some of the key parameters that would impact performance of an OC. Finally, an overview of JXTA is given in the appendix.

2 Previous work

This section will be dedicated to an overview of measurement techniques employed in contexts closely related to P2P Computing. Section 2.1 discusses the general philosophy of microbenchmarking. Section 2.2 focuses on measurements employed in clusters of worksta-

tions, a tightly coupled system, and the Grid, which can be seen as a more loosely coupled parallel platform. Section 2.3 describes a set of measurement tools and techniques used to gather information on the internet connection among a set of IP-addressed computers. Finally, Section 2.4 reports on the existing approaches employed to measure P2P system parameters and on the use of such information to improve P2P system performance, many of such systems being file sharing applications rather than computing facilities.

2.1 The microbenchmarking approach

When faced with the problem of determining the performance of a computing platform the major issue becomes the identification of the parameters that have the greatest impact on performance. To find such parameters, microbenchmarking techniques were first developed in a seminal paper by Saavedra et al. [41]. In that context, a microbenchmark is simply a small piece of executable code designed to exercise a specific hardware characteristic. For instance, the size of a cache and that of its lines can be determined by observing where discontinuities occur in the execution times of a program that scans many times an array of suitable size at varying strides. Similar approaches may be employed to measure associativity parameters of caches, CPU frequencies, pipeline depth of internal functional units, pipeline latencies, etc. A software package comprising several microbenchmarks is the *X-RAY* tool [51] developed at Cornell University.

Parameters measured through the microbenchmarks may be used to compare different computing architectures, or to compare the actual parameters against the ones advertised by the manufacturers. The technique may also be employed to obtain parameters to be used in a performance prediction model for individual or classes of applications with the aim of forecasting the performance of the application prior to actually running it. For instance, a microbenchmark suite may first estimate the size and the number of level of caches in the target architecture, and then use that information to instantiate the coefficients of a cost model for *a specific* application. The model can then be used to fit the application on the architecture [52] or to optimize the scheduling of the work [4].

When targeting *classes* of applications, the microbenchmarking approach can be applied with a *programming model* in mind. In this case a microbenchmark may be used to tune the parameters of a programming model, as was done in [1] and [5]. These works focus on the Bulk Synchronous Parallel (BSP) [50] model of computation which needs bandwidth and latency parameters of the platform's interconnection network to forecast the execution times of applications written under its framework. A suite of benchmarks are then used to collect execution times of several carefully chosen parameterized communications to extrapolate (by means of interpolation) the architectural BSP parameters and then estimate execution times of applications.

Microbenchmarking is typically implemented with the intent of measuring parameters as accurately as possible, hence it may be a time-consuming activity. To filter out noise in measurements, due to interference with the operating system, other user processes,

spurious network traffic, etc., each test may need to be run hundreds or thousands of times. In a setting where the resource scenario does not change over time this may be feasible, but if the resources vary unpredictably over time, as is the case in a P2P-based OC, this may limit the application of the existing techniques and call for the development of more flexible approaches that take into account such variability and are able exhibit tradeoffs between the computational requirements of the microbenchmarking activity and the resulting accuracy of its predictions.

2.2 Measurement techniques for Cluster-based parallel machines and the Grid

Computer clusters are used as a scalable parallel architecture. They are generally built using stand-alone computing nodes as building blocks connected by some interconnection network. Performance ranges from low-end Beowulfs made by off-the-shelf PCs connected through a commodity network, to high-end systems made by clusters of Symmetric Multiprocessors (SMPs) connected through specialized high-performance switches. The employment of the Message Passing Interface (MPI), which is the *de facto* standard for the development of parallel applications, allows these architectures to be seen as a homogeneous class.

While questioning whether the above classification is significant is beyond the scope of this report, it is important to observe that several efforts have been done to provide accurate performance measurement tools of the MPI programming model on these architectures. In our opinion it is useful to review such tools in order to get some hints on what can be accomplished using similar techniques in the OC setting. More specifically, we will analyze here a couple of works that address the problem of obtaining accurate, significant and reproducible measurements on clusters.

In [17], describing `mpptest`, a measurement tool for MPI over a cluster of computers, the perils of badly designed tests are highlighted. Here we report a synthesis of pitfalls to be avoided, in order to provide guidelines for our task, even though not all of them are significant in the OC context. The perils indicated in [17] are listed below, where we have highlighted (using boldface) those that are more relevant for our setting:

1. **Forgetting to account for the initial communication link**, since some systems dynamically create channels, the first communication may take much longer than subsequent communications.
2. **Ignoring contention with unrelated applications or jobs**, which can deteriorate computation and communication performance.
3. **Ignoring non-blocking calls**, since they can allow computation and communication to overlap but also may be essential for program correctness.
4. **Ignoring overlap of communication and computation** and thus underestimating achievable performance.

5. **Confusing total bandwidth with point-to-point bandwidth** since point-to-point may definitely be too optimistic in real settings.
6. **Comparing CPU time to elapsed time** since CPU time may not include the time spent in waiting for data movements between remote processes. In general, choosing the right time to measure is a fundamental issue.
7. *Ignoring correctness* since systems that fail on long messages may have an unfair advantage for short messages.
8. *Timing events that are short relative to the resolution of the clocks* which may incur in high measurements errors. A related error is to subtract the clock overhead from the measurements, which generally inflates performance estimations.
9. *Ignoring cache effects* such as cold starts or invalidation effects that a data exchange may incur.
10. **Using communication patterns different from those arising in the application**, since different patterns may exhibit very different execution times.
11. **Measuring with just two processors** which may lead to significant degradation in performance in real settings, for instance, if a receiver would poll on the number of different sources.
12. **Measuring with a single communication pattern** to grasp the overall inter-connection network behavior.

The principal statement of the paper is that a measurement must be reproducible, returning always the same result, while running an application several times may lead to very different execution times. To overcome this problem the authors state that the only reproducible measure is the minimum time among several measures, even though they provide the user also with the average and the maximum. A related problem is how to decide the number of tests necessary to obtain a significant measure with a good confidence. The choice on the number of tests to be taken is left to the user that can set up a parameter to establish a tradeoff between accuracy and measuring time.

Another issue is the message length to employ during measurements. Messages of different sizes exhibit different performance behaviors since, as different thresholds are crossed, different aspects of the architecture have an impact performance. The choice made by `mpptest` is to adaptively select the sizes in order to fit a curve with a minimal number of tests. It is useful to plot the measurements as the size of the messages vary, but this may be of a little help when a measure needs to be used to take instantaneous decisions. The paper also describes several tricks used to avoid the perils listed above, for instance the authors artificially introduce a fictitious computation of variable length to measure communication and computation overlap capabilities of the inquired platform.

Another interesting treatment of the measuring issues in cluster-based parallel computers can be found in [18]. The paper describes the *MPIBench* tool, another tool to provide performance parameters of parallel architectures based on MPI, which addresses the problem of obtaining accurate measurements without incurring in long testing sessions. The idea is based on the use of high precision timers available in today computers. By measuring start end ending time of each message-passing event at both sides of a communication it is possible to infer with good precision several parameters of the message exchange, such as flying time, setup time, etc. This is only possible if the system owns a global precise clock. To allow for the clock to be globally available, a synchronization algorithm is developed and a technique to filter the linear drift of local clocks is employed. The proposed method strictly relies on the availability of a global and very precise clock to allow for fast and reliable measurements of communication performance. In a P2P setting we will be forced to take measurements without the availability of a global clock, which would require substantial efforts to be implemented and whose precision would likely not be sufficient for the purpose.

Another platform for parallel and distributed computing is the Grid. Since scheduling decisions on the Grid are based essentially on performance declarations of the (trusted) institutions participating in Grid initiatives, there are few works that actually investigate the use of microbenchmarks to characterize the performance of applications running on a Grid-based platform. However, it is widely recognized that automatic benchmarking of performance would alleviate the work of system administrators and may avoid recourse to official declarations of capabilities of shared resources.

In [49] these issues are addressed and the *GridBench* suite is described. More interestingly to our purposes, in this paper the minimization of testing time is stressed, along with the reliability of obtained results. Attention is placed on the architecture of the system for the distribution of performance information, and on monitoring the state of a resource to validate the results. More specifically, the benchmarks are placed into a repository and executed periodically or on demand by the *Orchestrator* that also collects results and stores them in an archive accessible by the users. Once again, the tests are designed to use MPI as the communication layer. The test measures are the following:

- CPU performance on several kinds of operations. Each test is executed for a limited amount of time, typically less than 10 seconds.
- FLOPS and integer operation performance. The tests try to maximize the usage of the CPU's internal registers so to avoid the overheads induced by the memory hierarchy.
- Memory capacity test. The test repeatedly allocates exponentially growing buffers until the allocation fails.
- Memory bandwidth tests. The tests are based on loops performing operations such as copy, scale, sum, triad on arrays, running either a process per computing node or

one process per CPU to evaluate the performance degradation whenever multi-CPU computing nodes are used.

- Cache tests to discover the performance of the memory hierarchy system.
- Communication tests to measure latency, point-to-point, and bisection bandwidth.
- MPI I/O tests.

It is important to note that these tests are generally written in C and the submission of such programs to the scheduler is subjected to the policy of the grid. Moreover, the tests are intended to measure the performance of a given resource of the Grid and not to infer the global capabilities of the Grid as a distributed architecture.

Another work based on the Grid is *XtremWeb* [12] which addresses the problem of *Global Computing*, a framework where a program can use available computing resources around the Internet (*a la* Condor [48]) which is definitely more related to our target P2P environment for the design of an OC. The goals of the design are scalability, heterogeneity, availability, fault tolerance, security, and dynamicity. It has to be observed that the targeted systems are not fully P2P but rather descend from the grid approach. For instance, there are some XtremWeb servers, known to all the community, and the users are classified based on their efforts. Moreover, only trusted “peers” can submit jobs, while regular users can only donate their idle cycles to the trusted users. The reference framework is therefore reminiscent of systems such as the SETI@Home-like project, where a user acts as a volunteer for offering its own computing resources to a given project. XtremWeb appears to be an intermediate approach between the Grid and the P2P scenario, but still closer to the former: in fact, the reference architecture features scalability, heterogeneity, fault tolerance, security, and dynamicity, while job submission is ruled by central servers that orchestrate scheduling and monitor the execution to identify faults and to reschedule jobs to other nodes. Moreover, XtremWeb scheduling decisions are still based on declared parameters, even though a history for each host is collected to possibly help the allocation process.

2.3 Measurement techniques for the Internet

The availability of high bandwidth over the Internet has made possible the development of several kinds of distributed services, many of them relying on performance metrics to achieve good performance. This has inspired numerous research projects aiming at getting performance parameters out of the Internet with the double objective of minimizing the testing time while avoiding to overload the network itself. Typically, the required measures on the Internet are latency and bandwidth measures. Latency is often referred to as *distance*, since the latency between two nodes is mainly dependent on the path that a message follows to reach its destination, and hence it is dependent on the topology of the network (assuming that the topology itself is not changing across a suitable span of time).

While the simplest way to measure the latency is measuring *ping time*, that is, the Round-Trip Time (RTT) taken by an echo request packet between two hosts, obtaining a good approximation of the distance may require a long time. Also, not all Internet nodes accept ping requests due to security reasons. Being latency distances dependent only on topology, many works try to characterize the Internet using fictitious coordinate systems to place nodes onto a space, be it Euclidean or not. The principles of this approach can be found in [13, 35, 20]. There, the coordinates are evaluated by measuring distances (ping times) from some special hosts called *landmarks*, which operate as spots of known coordinates. Then nodes triangulate to find their own position onto the network. The major drawback in this approach is that special servers are needed (which require special administration and have stricter bounds on fault tolerance) which are not actually peers. Some mathematical properties of Internet coordinates are studied in [30]. An approach that uses unconscious landmarks is described in the following Section.

The aim of synthetic coordinates is to allow the estimation of RTT without prior communication, thus avoiding time-consuming probing. A synthetic coordinate system should provide:

1. Metrics that embed Internet with little error;
2. Scalability to a large number of hosts;
3. A decentralized implementation;
4. A minimal probing traffic;
5. Adaptivity to a changing environment.

Point 5 implicitly states that spending time in placing all the nodes into some space may be unsuitable, since it reduces the ability to adapt to the changes of the underlying topology. A well known coordinate system for the Internet that addresses all the five points listed above can be found in [8], where the *Vivaldi* system is presented. Interestingly for our reference scenario, Vivaldi is used in some actual P2P system implementations to improve the quality of the overlay network [40, 9].

Vivaldi's approach is inspired by the dynamics of a system of springs in a two dimensional Euclidean space, where the RTT between nodes i and j represents the elongation of the spring. The update algorithm is distributed. Each node computes the RTT with some other nodes and gets the coordinates of such nodes. With this information the node updates its own coordinates, minimizing a certain potential energy law which in turn automatically minimizes the quadratic error of the placement. The coordinate system provided by Vivaldi does not require any landmark and is totally distributed, hence it is suitable for the use by a P2P system. The algorithm lives as long as the system itself, and hence it continuously adapts to the changes in the topology.

The coordinates are all based on a common reference system, based on an initial random distribution of nodes. This may lead to some distortion of the space due to the

non Euclidean properties of the Internet. To overcome this issue, the developers of Vivaldi have introduced the concept of *Height Vectors*, that is, a third coordinate that specifies the height over the plane where the *springs* lie. The height coordinate models the overhead that each node experiences when communicating with other nodes. With the introduction of this coordinate the space is no longer Euclidean.

It is worth noting that a node needs only know the coordinates of a remote node in order to estimate the latency between them. Moreover, the communication to update a node's coordinates is only among direct neighbors of that node, hence the system results to be scalable and does not require too much probing traffic, since coordinate information may be exchanged during the regular operation of the system. Thus, all the previous desiderata for a coordinates system seem to be satisfied by Vivaldi.

It appears that measuring communication bandwidth is more difficult than estimating latency, since the bandwidth available between two hosts depends on several (global) aspects, such as the quality of the path connecting the two nodes, the amount of competing traffic along that path, the load at the end-points, etc. In other words, the bandwidth available for a communication depends on the overall state of the network during the operation. Since the load of the network exhibits a certain amount of regularity in its variability (for instance, the traffic varies during the day in an almost regular way), *instantaneous* measures of bandwidth could still produce results which are valid for a certain time interval. Estimations of bandwidth may be accomplished by exchanging data between two nodes and then dividing by the time spent in the exchange. Observe that this implies that the remote node is aware of the measure being taken and accepts to participate, which is a nontrivial requirement in the context of an OC.

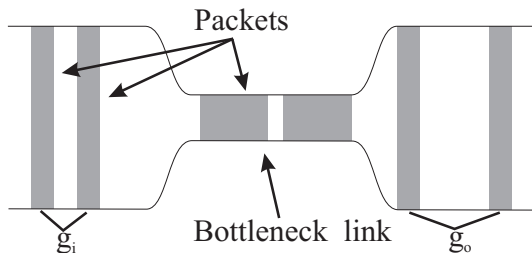


Figure 1: The bottleneck link will introduce delay in packets delivery that are related to its bandwidth. The size of the messages and the initial gap g_i needs to be set properly.

To get point-to-point bandwidth measurements as much independent as possible from the incidental conditions of the network, some techniques have been developed to measure, for instance, the *bottleneck link bandwidth* using the *packet pair* (or *packet train*) strategy, the bottleneck link being the link along a path with the smallest bandwidth capacity. Descriptions of such techniques can be found in seminal works such as [6, 27, 32]. In [6] the idea of a packet pair is described. A packet-pair is a pair of messages of equal size S sent a specified time apart, called *input gap* g_i . When these two packets reach the bottleneck link, they will be enqueued and possibly delayed. By measuring the dilation

(called *output gap* g_o) at the destination, it is possible to infer the available bandwidth on such a link (as S/g_o) (see Figure 1). The approach uses ICMP messages of different sizes to compute the g_o parameter at the sender side, so to avoid the remote node to be aware of the ongoing measurement.

In [43], the packet-pair technique is employed in uncooperative settings like P2P systems. While previous works were only able to determine the bottleneck bandwidth along the direction from the inquiring node to the remote node, here the authors implemented a strategy to determine the reverse bandwidth of the path going from the remote node to the inquiring node. To do so, the presence of a web server is required in the remote node. The inquiring node sends two packets requesting data from the server, advertising a 1500 byte Maximum Segment Size. The server then answers by sending one or two segments, depending on the size of the congestion window of the server. If the congestion window has length 1, the inquiring node timeouts and acknowledges the server, which increases the size of the congestion window, thus allowing the packet pair to be handled correctly. This measure is considerably more unstable than the forward direction bandwidth, but the author states that 85% of the measures were within a factor 2 from the actual bandwidth.

The above is an example of how a clever technique may exploit the characteristics of the underlying protocols. Another example of this microbenchmarking approach can be found in [44], where an interesting methodology to measure the quality of a connection among two nodes had been proposed. In this paper, in order to probe a remote node, a node sends to it some TCP packets ignoring the acknowledgments. At the end, since the remote node acknowledges with the sequence number of the first packet it lacks, the node starts filling the holes generated by previous blind sends. With some precaution to avoid to account for acknowledgement loss, it is possible to quantify the loss rate of the path from a node to another. This, however, is highly protocol-dependent and hardly applicable to a P2P system.

More recent works [22, 21] improve the packet-pair technique by providing better filtering of the results and algorithms for automatic selection of the initial gap. In [27] similar techniques are employed in a tool reminiscent of *traceroute*, called *Pathchar*. Pathchar, like traceroute, exploits the fact that a router must identify itself when the *Time To Live* (TTL) parameter of an IP packet reaches zero. Then, by sending a packet-pair, or more robustly a packet-train, with an increasing TTL parameter it is possible to spot the bottleneck link along the path and to find the bottleneck link bandwidth toward any router in the Internet. A similar technique is used in a more advanced tool described in [24].

As stated before, while coordinate-based approaches allow latency estimation without explicit communication, an analogous approach is more difficult for bandwidth estimation. More specifically it is difficult to answer the following question: given an N node system, what is the available bandwidth in the system? The work described in [24] tries to answer the following question: Given an N node system, is it possible to evaluate all the $O(N^2)$ point-to-point bandwidths with less than $O(N^2)$ measurements? Considering that in order to probe a single point-to-point path around 100KB are needed, probing a 150 nodes

system would entail an exchange of more than 2GB to perform full measurements. The authors employ a quite complex approach to reduce the number of measurements. The intuition is that bottleneck links are typically close to path end-points [26] and that a limited amount of information per node (constant with respect to the size of the system) can be used to infer bottleneck link bandwidths with high probability.

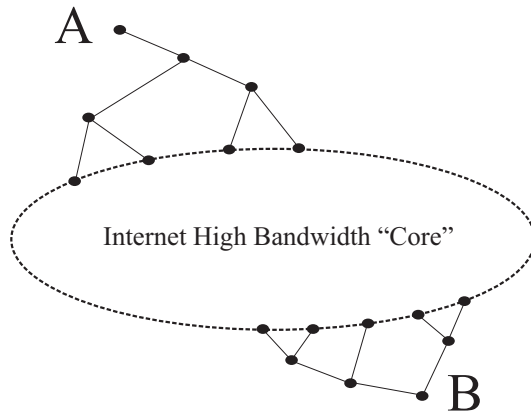


Figure 2: If the bottleneck link is located near one the end-points of the path from A to B , then it must be on the trees that connect A and B to the high-speed backbone of the Internet.

The hypotheses upon which the work is based are the following: first, the Internet is built by autonomous systems (ASs) organized in *Tiers*, where different Tiers have a customer-to-provider relationship, or they have a peer/sibling relationship. Therefore, to go from a source to a destination, a path goes typically along a tree and traverses different tier levels. Secondly, the bottleneck link is at a constant number of edges from the end-point. If a node computes the bottleneck link bandwidth along all the paths of constant length E outgoing from it, it then can store such an information to be used by other nodes to query for bottleneck link bandwidth. For instance, if node A wishes to know the bottleneck link bandwidth to a node B , it needs to know the first and the last E edges of the path. Then it takes the minimum bottleneck bandwidth on these path to obtain, with high probability, the bottleneck link bandwidth from A to B . Figure 2 contains an example of such an approach. A dedicated server (but an advertising approach would also work) will receive the queries and find the match between paths, returning the available bandwidth.

The amount of communication to provide such a service is proportional to $N \times E$, where N is the size of the system and E is the (constant) depth of the trees that are supposed to contain the bottleneck link. Using this system it then possible to infer bandwidth without previous communication, as for the coordinate systems used to infer latency. However, the infrastructure to be provided is quite complex and the hypotheses upon which the whole approach relies may be difficult to satisfy in the context of an OC.

2.4 Topology awareness techniques for P2P Systems

P2P systems, as large and geographically distributed systems, employ several techniques to provide better performance by exploiting topological characteristics of the underlying networks. *Topology-awareness* affects the scalability of such systems as discussed in [53], where the authors indicate two main critical aspects: the core algorithm for managing routing tables and the caching capabilities needed to provide locality of reference. Since at the time the paper was written there was little availability of large scale systems to provide experimental feedback, the paper contains a mainly qualitative analysis of the key aspects that highly impact scalability in topology-aware systems.

Recent works have attempted at a more quantitative analysis. The work in [9] reports on an extensive experimentation of *DHash++*, which is an implementation of a Distributed Hash Table (DHT) [3] based on *Chord* [47]. Namely, the authors experiment with several different lookup algorithms under many assumptions. The performance of *DHash++* is then studied from the point of view of both lookup time and throughput. The lookup time is highly affected by locality-aware implementation of protocols, which can halve the time of the average request. Performance improvements are obtained by refining the lookup and fetch algorithms to include features such as recursive lookup, proximity selection, server selection, and the integration of lookup and data fetch. Throughput is also found to be locality-aware, but only latency is taken into account as a measure of distance. The time to transfer data has not been accounted for, since the data to be retrieved is only 8KB long in the application under examination. *DHash++* employs Vivaldi [8] to estimate distances. The lookup protocol exploits Vivaldi to identify nearby nodes while throughput optimization uses Vivaldi to compute proper timeouts depending on the destination of a message. The results are based on tests run on PlanetLab and RON platforms and on simulations on the King data-set [20].

Earlier works also tried to characterize P2P filesharing application workloads in order to evaluate the improvements that topology-awareness would provide if included in the implementation. One such work is [19], which analyzes the traces of Kazaa traffic within the network of the University of Washington. The traces are taken over a period of 203 days, during which more than 1.5M requests were monitored for a total of 100M transactions (other data are available in the paper). The paper uses the traces to give a quantitative characterization of the users's downloading habits, workload, popularity of content, etc. One interesting peculiarity of filesharing systems is that a user downloads a given content just once, while Web users access the same content several times. With such a consideration in mind, the authors simulate the effects of a caching system whose intent is reducing the traffic outside the University of Washington. Their results suggest that, if content were downloaded exclusively within the University (after an initial cold-miss), then about 60% of external bandwidth (between University and the rest of the world) would be saved. The analysis is based on some optimistic assumptions (for instance infinite peer's storage) and some pessimistic assumptions (a very conservative view of a

peer’s availability). Other hypotheses, as the available bandwidth per peer, were based on average figures. More by the same authors on the analysis of traces of P2P applications, can be found in [42].

Another paper that analyzes traces in order to measure characteristics of a P2P architecture is [31]. The report presents *PeerMetric*, an effort to measure P2P network performance from the vantage point of view of the broadband residential hosts. The final conclusion of the paper is that last-hop bandwidth is a major bottleneck and that latency-based optimization of the overlay embedding is somewhat a less impacting issue. Moreover, the authors claim that latency is a poor indicator for throughput, and that bandwidth measures may be of interest for P2P (bandwidth intensive) applications.

Measures were taken from 25 broadband hosts. The monitored traffic was not only generated by P2P applications, but the authors use the overall traffic as an indicator of the P2P traffic. They collected TCP throughput, ping, packet pairs, and traceroute measurements during a 25 days period by means of individual agents running on each host. To summarize, we list the main conclusions of the report:

- Upstream and downstream bandwidth are highly asymmetric, with upstream bandwidth being the bottleneck;
- P2P latencies are higher than those between well connected hosts by an order of magnitude;
- Ping time is a poor indicator for throughput;
- Coordinate-based peer selection is a profitable technique;

One of the firsts implementations of a topology-aware P2P system is *Pastry* [40]. In Pastry, a node comes with a random ID that identifies the position of the node on the overlay topology. To allow for locality optimization, a node can choose the nearest among k potential neighbors (nodes whose ID is close to the node’s ID). The proximity metric chosen by Pastry, instead of being the *latency distance* (i.e., the ping time), is the number of Internet hops in the path between two nodes. The selection of the closest neighbors is not among all the nodes participating in the system, so it is intrinsically sub-optimal. The number of IP hops is a metric rarely used in other works. If the latency is dominated by the routers’ latency, this metric may be used as an approximation of latency time, with the advantage that number of hops is a steadier quantity than RTT time. The measure of the distance is however delegated to external programs, such as `traceroute`, and no efforts have been made by the developers to provide a more specific measurement system.

While Pastry was the first system to feature topology-awareness, other systems were subsequently improved to include such a capability. An example is provided by CAN [37]. In CAN the ID space is represented by a d -dimensional unitary cube in the ordinary Euclidean space. A peer in CAN is assigned a tuple of coordinates that identify the peer as a point within the cube, and the peer becomes responsible for a rectangular d -dimensional

region of the cube. An item is also associated with a point within the cube and stored into the peer associated with the region where that point lays. When a node wants to join CAN, a large region is found and bisected. The new node is assigned half of the region and the peer that was previously responsible for the entire region will take the other half. The list of neighbors of the peers around the new region is then updated and data are redistributed accordingly. The only rule that drives the insertion of a node is to improve load balancing by bisecting large regions.

The work presented in [38] proposes a strategy to insert topology-awareness into CAN. The idea is however more general and the authors state that it can be also applied to unstructured P2P systems (where the placement of data items is not decided through hashing), and to the problem of server selection over the Internet. In the proposed method, the node set is partitioned into *bins*. The task of a node that wants to join the system is to find a suitable bin where to fall. Then a node probes a set of predefined landmarks in order to derive a *bin identifier*, that is, the list of the landmarks sorted by increasing distance (latency time). By looking for other nodes with the same bin identifier, performance can be improved by bisecting a large region among those associated with peers already in the bin. In unstructured P2P systems a node has the possibility to select neighbors with more freedom than in structured ones. The proposed method may help the routing algorithm to select better neighbors also in the unstructured case. For server selection, the technique helps the identification of the nearest server that can provide a given content. The presented results show that topology-awareness can significantly help in improving performance (measured in terms of latency stretch) by a significant factor that depends on the topology of the underlying network.

Binning is actually a little more complex than sorting landmarks by increasing distance. The distances themselves are binned and the number of landmarks at a given distance are appended to the list of landmarks to improve bin selection. Another interesting feature of this method is that a landmark may not be aware of its landmark status. A landmark may be a DNS server, or a web server, etc. They are chosen at random with the only constraint to be some number of hops apart from one other. Of course, the temporary crashing or the dismissal of a landmark needs special care.

Among other efforts to get topology-awareness in structured P2P systems, the work in [36] discusses a strategy to make a general P2P structured network latency-aware. The nodes of the overlay network perform random probes to discover neighbors and then simulate the swap of IDs. If the swap leads to an improvement in overall latency, then the two IDs are actually swapped (possibly along with the nodes' contents). More precisely, a node *A* probes a node *B* and the neighbors of that node, and computes the overall latency improvement in the case *A* and *B* swapped their identifiers (and hence their placement in the overlay network). The results, obtained by means of simulation from datasets, suggests that this approach can lead to improving the latency stretch of about 40%.

The authors take into account anonymity issues, since IDs change over time, and employ several techniques to improve the performance of a single adaptation, since a swap

tends to be quite a heavy operation. To reduce the impact of swapping, they choose to probe the network using an adaptive strategy, that reduces the frequency of probing as the system becomes steadier. From our perspective, it is important to note that once again a measure of latency (and not bandwidth) is assumed as a measure of distance and that the system needs to probe continually during its lifetime to maintain its topology-aware characteristics.

While structured P2P systems, like DHT, exhibit good proprieties in terms of routing performance, anonymity, fault tolerance, etc., they present the drawback that general queries to search for a specific content are not handled efficiently. If a user needs a content it must provide the exact name of such a content to be hashed. Systems like Gnutella use an unstructured approach that allows for general queries to be matched partially, and present the user with a ranked list of possible contents. In Gnutella, a peer is connected with a number of other peers. When a query is injected into the system, it will be propagated in a breath-first manner. Each query has a time-to-live (TTL) parameter that is decreased by one at every hop. When TTL drops to zero the query is no longer forwarded. To avoid unnecessary flooding, a peer checks if the same query is already pending before forwarding. When the query finds a match in a peer a positive answer is returned to the initiator of the query, together with the content found.

In [7], a method to make Gnutella-like systems topology aware is presented. The paper advocates for a Gnutella-like design (called GIA) against the DHT approach. GIA adapts itself to the underlying network and to the peer's *capacity*. Capacity is the number of queries that a peer can process without being overloaded in some of its components: bandwidth, computing power, disk latency, etc. Lookup protocols are then designed to guarantee an even utilization of the capacities of the peers involved. GIA relies on four components:

- *Topology adaptation* to allow high-capacity peers to handle a higher number of queries and to place low-capacity peers at short distance from high-capacity ones. The algorithm employed is based on a *satisfaction level*. A peer continues to adapt until its level of satisfaction is high enough;
- *Flow control*: To avoid hot-spots, a receiver accepts requests only from peers that were previously allowed to issue requests. This is implemented using tokens. A peer can issue a request only when it has the token;
- *One-hop replication* of pointer to content: Each GIA node maintains the index of the contents of each of its neighbors. When a neighbor disappears (either because it left the network or it is no longer a neighbor due to adaptation), its index will be flushed;
- *Search protocol*: search relies on a *biased random walk*, where the next node in the path is chosen among the highest capacity neighbors for which the peer has a token.

The results indicate that the adaptation improves performance by orders of magnitudes with respect to the basic Gnutella system. From the point of view of this report, it is important to note that this is one of the few papers that also take into account the computational power of the peer, along with the traditional latency and bandwidth parameters. A similar parameter is considered in [29].

The idea to organize the network with emphasis on high-capacity nodes is also adopted in [46]. The paper describes a method to build a hierarchical overlay topology where nodes are classified in terms of their capacity. Nodes with the same capacity are considered to be at the same *Heterogeneity Level (HL)*, where a low level is bad. A node at a given *HL* may be connected only *upwards* to a limited number of nodes at level $HL + 1$, or *downwards* to a number of nodes at level $HL - 1$. Nodes at the same level may be also connected among themselves.

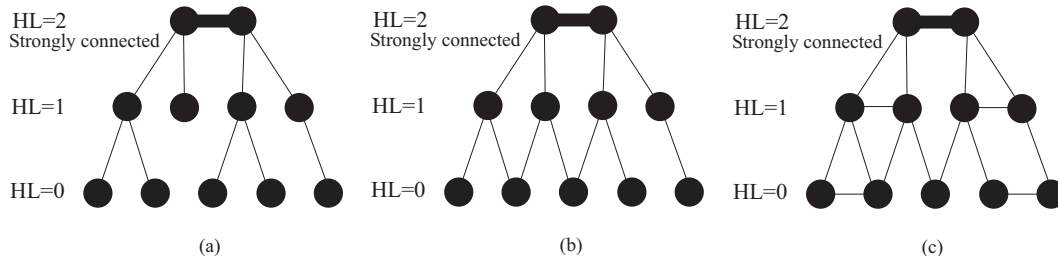


Figure 3: The three kinds of tier topologies outlined in [46].

There is a hierarchy of topologies based on the degree of fault tolerance that each of them can exhibit. Three kinds of topologies are considered:

- *Hierarchical Topology*, where only the nodes at the highest level can be connected among themselves while the others are arranged in a tree where nodes with higher HL are closer to the root (Figure 3.a);
- *Sparse Topology*, where a node at level HL may be connected to more than one node at level $HL + 1$ (Figure 3.b);
- *Dense Topology*, where nodes at any given level may be connected among themselves (Figure 3.c).

The nodes at the highest level form a *backbone* structure that guarantees high availability. The resulting topology is essentially a tree (with extra edges for fault tolerance), where the levels near the root have more bandwidth than the nodes at lower levels (a structure reminiscent of a *fat-tree* organization [33]). The paper focuses on adaptivity with respect to the routing algorithm which has to select the best neighbors to propagate queries. Unfortunately, in the paper the notion of capacity is not properly quantified: it is only stated that its value should be related to the available bandwidth and computing power of a node. The results indicate that substantial bandwidth savings can be attained

when employing one of the topologies listed above with respect to a random topology. Noticeable gains can also be obtained in load distribution, since the heaviest load goes where it can be properly handled. Latency is also reduced. From the point of view of fault-tolerance, only the Dense Topology guarantees a substantial improvement over the random topology, while the others are only slightly better.

Other aspects impact the overall performance of a P2P system, for instance, the *churn*. The churn is the rate at which peers join and leaves the system. When a node joins or leaves, other nodes in the network should perform some tasks to maintain the consistency of the system itself. This phenomenon may affect the performance of the routing, especially if time-outs are not sized properly. When a peer wishes to contact another peer, it sets a timeout after which it may consider the peer not available. If the time-out is determined without any consideration on the actual distance of the peer (if for example it is chosen based on the diameter of the network), it may be the case that a peer waits longer than necessary to decree the unavailability of another peer.

The work in [46] includes a description of Bamboo, a DHT that explicitly addresses the problem of routing performance under heavy churn. Bamboo nodes route messages through two data structures: the *leaf set* and the *routing table*. The leaf nodes are chosen to be nearby (in terms of latency) to the node itself, and the routing table entries, when multiple choices are available, are also based on distance criteria (like Pastry [40]). When a node fails, the rearrangement happens according to distance criteria. Bamboo employs Vivaldi [8] to get distance information without prior communication and therefore set up reasonable time-outs. The opportunity of choosing nearby peers is critically weighted. The goal of leaf sets is to guarantee correctness (even though it requires $O(N)$ hops, where N is the number of nodes in the network), while the routing table allows a fast, $O(\log(N))$ -hop routing. To be able to fill the routing table as quickly as possible, instead of finding the nearest neighbor, a faster but coarser method to find a few close neighbors is employed. Moreover, since the neighbors change during time due to churn, the authors state that it is not so urgent to find the nearest neighbor to get substantial performance improvement. Hence, they start filling the routing tables with little effort for locality and then refine the choices periodically.

Results indicate that the proximity approach attains a lower latency under high churn. It is also highlighted that the best recovery strategies to be employed when a node joins or leaves depends on the churn level. For instance, a periodic recovery facility is able to guarantee lower bandwidth consumption with respect to a reactive strategy that acts as a node joins or leaves in case of high churn.

Few works try to face the problem of topology-awareness for P2P computing (rather than filesharing) systems, since there are not yet many such platforms. A paper that describes a P2P architecture for supercomputing that is also topology-aware can be found in [10]. The paper begins by describing the following features that the job scheduling algorithm of a P2P computer needs to provide. Such an algorithm needs to:

- Be distributed;
- Support co-allocation (simultaneous allocation of resources);
- Be locality aware;
- Adapt to a quickly changing environment;

The starting hypothesis is that all such features cannot be provided by a Grid infrastructure, which is difficult to manage and mainly centralized.

The authors then describes Zorilla, a prototype P2P supercomputing platform that addresses the issues described above. An algorithm similar to the one implemented in Gnutella is used by Zorilla for discovering peers when allocating jobs. The algorithm has been modified to be locality aware in a way similar to the strategy employed in [7]. The result is that the reached nodes are close to the node that initiates the job submission, rather than being randomly distributed as they are in Gnutella. As stated by the authors, this feature speeds-up the initial phase of moving input files from the submitting node to the workers. While it may be reasonable, it looks like a limitation to force the selected peers to be closer to the origin rather than simply to one another (consider, e.g., the case of computations not featuring heavy I/O).

The scheduling algorithm automatically sets a radius for constraining the search of peers in the neighborhood of the initiating peer. Locality awareness is guaranteed by the topology-awareness of the overlay topology. Zorilla's peers that are close in the overlay topology are also close in the physical topology (latency is taken as a measure of distance). The overlay network is built using Bamboo [39], which provides the level of locality awareness needed by Zorilla.

Zorilla is a tentative framework for P2P supercomputing. Topology-awareness is addressed and this is a step toward a performance-conscious P2P computing system.

3 Requirements for the Overlay Computer

An overlay computer should provide functionalities to make the computational power of the single participants available for the execution of a number of applications. However, it is also necessary that the overall performance achievable by the coordinated effort of a number of hosts be a significant improvement with respect to the computational resources available to the user itself. Only in this case would a user be encouraged to participate to the overlay computer and share his/her own resources. If the improvement is measured in terms of running time, then a user should be provided with evidence or tools to establish if running an application over the OC (rather than locally) is profitable. If the improvement is in the total memory available in the OC, then the user may choose to participate also if computational times are not as competitive. In any case, in order to assist the user in the decision on whether and when to resort to the OC for running his/her applications, one needs ultimately to provide measurements techniques for the OC.

Most of the current parallel applications running on P2P systems are compute-intensive and embarrassingly parallel [45, 16]. These are in the class of *infinite workpile* applications, as characterized in [34], which means that there are no requirements on the time the computation must terminate. The computation graph of these applications essentially consists of client-server relations. When running more complex applications with a certain degree of interaction between the participating entities, the challenge is to guarantee that communication and computation performance be sufficient to obtain some speed-up.

Since it would be difficult to provide such a speed-up for a generic parallel application, it may be useful to resort to a classification of applications based on their requirements and characteristics. A preliminary classification can be found in [34], where along with the infinite workpile one, other application classes are introduced, namely *workpile with deadlines*, *tree-based search*, *point of presence*, and *tightly coupled parallel* applications.

In our opinion, a finer classification is needed in the context of overlay computing. For instance, tightly coupled parallel applications represent a broad class containing applications with very different characteristics and requirements. A more useful classification of the applications should include more specific aspects regarding computational resources, such as the exhibited communication patterns and computing power requirements. As an elegant way to encapsulate these properties one can resort to the BSP model of computation, briefly discussed in section 2, which provides a programming model able to decouple computation from communication. Works like [1] and [5] showed that different communication patterns exhibit very different performance, which is an indication that classifying applications in terms of communication patterns most frequently used would lead to a more effective performance-conscious setting for the Overlay Computer. As an example, an algorithm relying on *scatter communications* is best suited for architectures where the initiator has a great upload bandwidth than an algorithm based on *gather operations*. the suitability of BSP as a parallel programming environment for the OC computer is also one research line within the AEOLUS project.

From the material presented in the previous sections we envision the two following fundamental tasks for the OC that could receive a decisive help from the availability of a suitable suite of measurement facilities. The first being *topology optimization*, that, as seen in the previous sections, efforts to improve topology-awareness pays off in terms of the overall performance gain. Many works in literature focus on ping distances to reduce look-up time, while few of them also consider other measures, such as *capacity*, a quantity that takes into account both the available bandwidth and the host's computational power.

As peers join the OC, decisions about where to place them, accordingly to topology-based objectives and the characteristics of the peer itself, would be taken by considering performance metrics for the sake of optimization. An example of such an approach may be seen in [46] which revolves around a FAT-tree-like architecture. Explicit measures of bandwidth and computing power are then necessary.

On the other hand, it may be necessary to state which computations can be efficiently carried out over the topology (or a portion of it). An *a posteriori* measure system should

then be devised to give a quantitative assessment of the features of the topology with respect to high level capabilities (such as the ability to route certain communications patterns, or the amount of computational resources available) so to provide the user with reasonably accurate insights on the achievable OC performance for a given task.

The second task involving performance measurements is *Application Side Optimizations*. A user needs to know if his/her application could benefit from running on the overlay computer. For instance, the user may want to query if the OC provides a certain service for that kind of application (say, a subnetwork with certain characteristics). What the user needs next is spawning the application over the network. The spawning strategy will depend on many factors, such as the policies to monitor the application, the responsibility of the user's node, etc. For instance, Zorilla [10] and P2P-MPI [14] both require that the spawner peer participate in the actual computation. This may force performance-conscious spawning to choose a set of processors "close" to the spawner, which may be a limitation if the peer does not participate in a sub-network suitable for the purposes of the application.

In general, any performance-conscious spawning strategy should provide a fast and reliable way to identify a set of peers that, according with the stated policies, maximize performance. This may be accomplished by executing direct performance measures or by looking up at previous performance statistics. Both approaches have points of strength and point of weakness. Direct measurements can capture the current state of the network and be the base for more precise spawning strategies. However, the approach is definitively resource consuming and unpractical for very large systems. For instance, a detailed characterization of the bandwidth in a large set of peers may require that large volumes of data be exchanged; probing computing power may require lots of CPU cycles to be spent, etc. Look-up may guarantee faster spawning decisions but may reflect a network state that is no longer valid. A mix of the two approaches could be a viable compromise, for example by employing bottleneck link bandwidth instead of current bandwidth and relying on some stability characteristics of the network as seen by a given user [11].

The requirements for topology and application optimizations are different. While the time spent for topology optimizations can be possibly long to allow for accurate measurements, the time spent in performance evaluation for efficient spawning of processes needs to be considerably shorter. As an instance, to measure bandwidth a precise method involving the exchange of relatively large blocks of data can be employed when optimizing topology, while a faster method based on smaller data (and then with ping-like measures) needs to be employed when optimizing the spawning of processes.

Similar considerations hold for measuring the computational power of a peer. Also in this case two methods may be employed: the first sets the amount of computation to be performed and then measures time, while the other sets the time to be spent in computing and then measures the number of operations, maybe obtaining coarser information due to caching effects [17]. The first technique is best suited for topology optimization, while the second is best suited for spawning optimization.

The next section describes a suite of benchmarks implemented over JXTA (see Appendix A) whose intent is to provide preliminary insights on performance characteristics to be exercised to obtain useful performance parameters.

4 Design and implementation of initial tests

This section describes a suite of experiments aimed at identifying the main characteristics that impact performance in a P2P system based on JXTA (see Appendix A). The tests have been implemented using the *JXTASocket* interface, which provides reliable bi-directional communications. Lower level interfaces (e.g., *Pipes*), although faster, have not been used because of their unreliability. Our first objective is to measure key performance quantities at the user level for tuning subsequent more specific benchmarking tools for the OC. To achieve this objective, tests have been designed for measuring *latency*, *bandwidth*, and *computing power*. We remark that the following experiments have to be considered as a *proof-of-concept* behind the development of a comprehensive benchmark suite.

To measure latency the core of the experiment is a simple *ping-test*, where a peer sends a small packet (8 bytes) to a selected peer, which then replies as soon as it receives the message. To filter out noise this process is iterated several times. The initiating peer then computes the round-trip time by averaging over the iterations. Since JXTA is based on Java, it is important to quantify the software overhead. For this reason we have measured ping-times between fast and slow computers and also compared against times obtained through the ICMP protocol.

Bandwidth is measured through several tests. Each peer involved in the benchmark measures the time for receiving and/or sending the amount of data assigned to it. It has to be remarked that identifying the end of the process may be problematic, since, for example a peer can easily determine when it finishes receiving data but has no direct control over the instant when it actually finishes sending them because lower layers of the protocol stack can possibly delay the actual data delivery. To avoid this problem, the test may require that a peer acknowledges the sender when all the data from it has been received. Sending time is then adjusted by subtracting the average RTT from a peer to the destination. With these precautions each peer can manage the measurement of time without incurring in additional communications or resorting to a precise clock alignment algorithm, which is hard to implement in a P2P setting. The bandwidth is then computed as the sum of the outgoing and incoming bytes divided by the measured time. This bandwidth measure captures both the capability of the peer and the state of congestion of the network. Bandwidth estimations of all the peers involved in a test are collected and the minimum, maximum, and average values are computed and then plotted as the amount of data exchanged varies. This approach provides a uniform measure of bandwidth and a clear way to compare results of different communication patterns.

When measuring bandwidth, a simple clock alignment algorithm is employed to ensure that peers start the measurement roughly at the same time. To perform clock alignment

a selected peer measures latencies from it to all other peers involved in the test, and then sends them the time they have to wait before starting the experiment. This algorithm is quite simple and is used in this preliminary experiments to provide a light weight synchronization of the peers.

The first measurement concerns **point-to-point** bandwidth and is implemented by making one peer send a given amount of data to another peer and wait for the acknowledgement from the receiver. An estimation of the bandwidth *towards* the destination is then obtained by dividing the amount of data exchanged by the communication time. Several message sizes have been employed to identify the point where the bandwidth saturates.

We also measure the execution times of **gather** (i.e., all-to-one) and **scatter** (i.e., one-to-all) communication patterns, to evaluate the OC communication capabilities with respect to typical patterns used in the applications. To measure the execution time of the gather pattern, the alignment algorithm is used to make all peers send the data to the *collector* at the same time. The collector uses a number of receiving threads equal to the number of peers sending data to it.

Similar considerations are valid for scatter: a *distributor* sends data to a number of involved peers and measures the elapsed time after receiving the last acknowledgement. It employs a number of threads equal to the number of receiving peers. All the bandwidth measurements have been carried out with different configurations involving fast and slow machines to provide insights over the software overheads.

A global measure of bandwidth could be provided by running an **all-to-all** communication pattern where every peer sends the same amount of data to all of the others. This pattern, which may prove less crucial of an OC since it requires the availability of a large bandwidth to be competitive with tightly coupled architectures, is not yet implemented, but will be included, for completeness, in the full benchmarking tool.

To measure the computing power of a remote peer we use a quiz-like approach [34]. The remote peer is required to perform a given computation and the execution time is measured by the inquiring peer (that needs to filter out communication time), since the it needs to trust the measure in a uncooperative P2P setting. In order to be effective, the computation needs to exhibit the following characteristics: 1) requiring both a small input and a small output; and 2) requiring a given amount of computation that cannot be avoided. The first requirement is necessary since we do not want the network to become a bottleneck, the second allows for a trusted measure, since the inquired peer cannot employ faster algorithms to provide the right answer. If the peer answers correctly it must have carried out the given computation, whereas if the answer is wrong either it cheated or something went wrong. A computation that matches the requirements is given by a random number generator where the input is the seed and the output is the number generated after a given number of iterations.

Computing power can be measured in two ways: the first is by sending to the peer the seed and the number of iterations to execute, the second is by sending the seed and

the time we want the test to run. In the first case the answer from the peer is simply the generated number. In the second case, the provided answer is the number generated and the number of iteration executed. In both cases the tester can check the correctness of the received answer. Using the test variant where the computational time is fixed ahead is preferable if either the measure is needed quickly or if the tester is not allowed to consume too many cycles on the remote machine.

5 Results

Cat.	CPU	Frequency	O.S.	Distribution
slow	Pentium III	800 MHz	GNU/Linux 2.6.16-	Debian Etch
slow	Pentium III	600 MHz	GNU/Linux 2.6.12	Vector Linux 5.1
slow	Pentium III	866 MHz	GNU/Linux 2.6.16	Slackware 10.2
slow	Pentium III	866 MHz	GNU/Linux 2.4.18	RedHat 7.3
fast	Dual Opteron	2.2 GHz	GNU/Linux 2.6.5 smp	Suse Linux 9
fast	Pentium Core Duo	1.66 GHz	Windows Xp sp2	-
fast	Pentium 4 HT	3 GHz	GNU/Linux 2.6.13 smp	Suse Linux 10

Figure 4: Description of computers used to perform experiments.

This Section shows some results from preliminary experiments on a network of 6 heterogeneous computers connected to a 100Mbit/s Ethernet switch. Test code is written in Java over JXTA and implements the benchmarks described in the previous section by measuring latency times, uni-directional bandwidth, gather and scatter bandwidth, and computing power. The machines employed during the experiments are described in Table 4, where each computer is identified as *fast* or *slow*. This distinction is necessary to evaluate the impact of the software layers on performances.

Sender	Receiver	Time (ms)	ICMP (ms)
Fast	Fast	17.2	0.24
Fast	Slow	70.4	0.16
Slow	Fast	57.6	0.14
Slow	Slow	85.5	0.22

Figure 5: At application level, pings are order of magnitude higher than the ICMP pings. Application level ping times also depend on the computer architectures involved.

In Table 5 the latency measurements are showed, by means of *ping* measures. The table reports two ping times, the first is the *application level ping*, where JXTA processes exchange short messages and the RTT is measured. The second time is the time obtained by running the ping command between pairs of machines. It possible to note that the JXTA-level ping is up to three order of magnitude slower than the ICMP one, because

of the software overhead introduced by JXTA, hence the faster times are obviously those among fast computers. It is also possible to note that the JXTA-ping time from slow to fast computer is lower than the one from fast to slow. A similar phenomenon is visible also in the uni-directional bandwidth measurement presented in Figure 6.

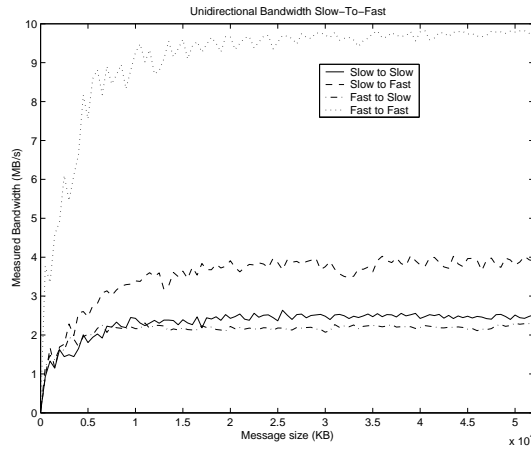


Figure 6: The unidirectional bandwidth exhibits a strong dependency on the computing power of the peers. Fast to fast data send is more than three times faster than configurations involving slow computers.

Unidirectional bandwidth (Figure 6) shows the high dependency from the underlying architecture. When employing fast computers the bandwidth is quite close to the peak bandwidth of the 100Mbit/s Ethernet (i.e., 12.5MB/s) while slow computers dramatically slowdown the communication speed. We note that fast-to-slow communication is faster than slow-to-fast. This phenomenon, also visible in latency measures, will be investigated in future research.

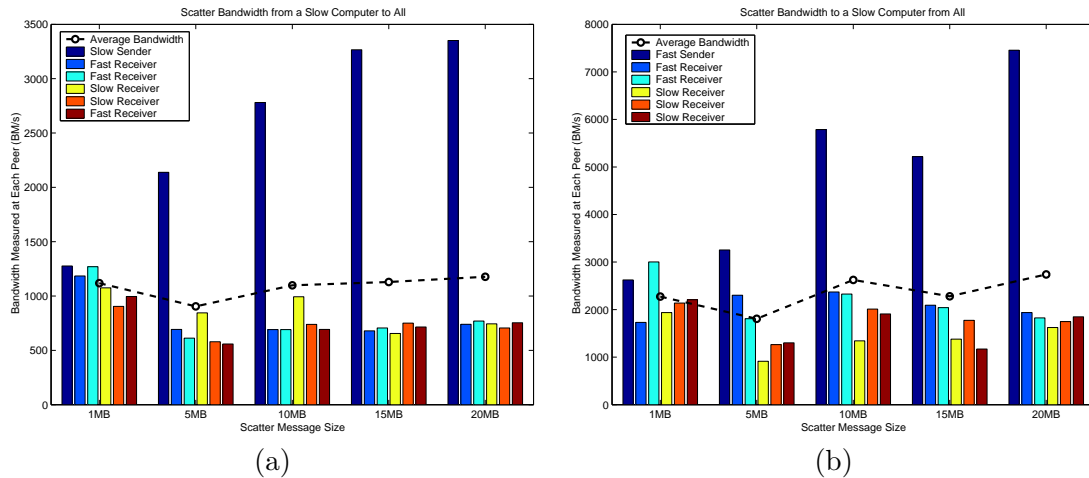


Figure 7: Scatter measures, respectively, from a slow computer to the others (Figure (a)), and from a fast computer to the others (Figure (b)). The plots show measured bandwidth at each peer along with the average value, varying the size of the message received by each peer.

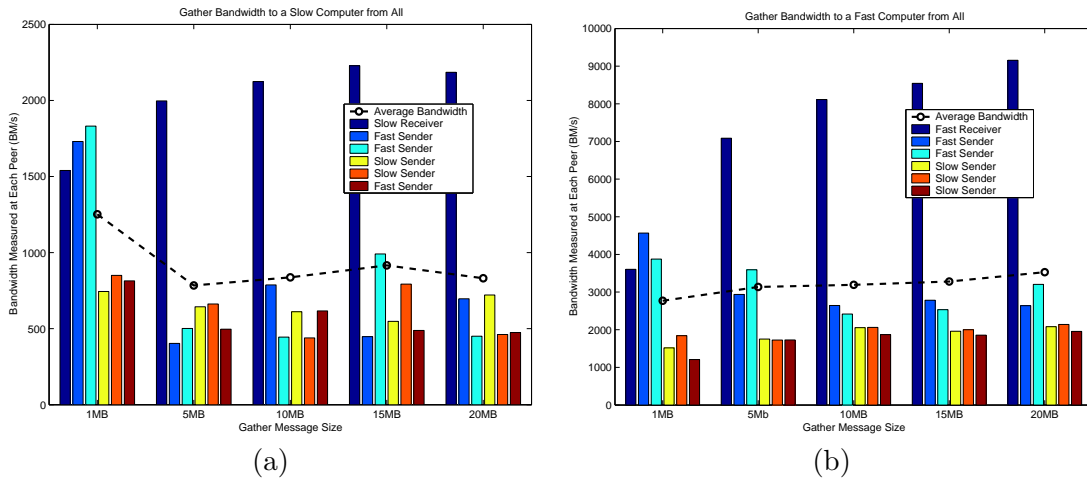


Figure 8: Gather measures, respectively, from a slow computer to the others (Figure (a)), and from a fast computer to the others (Figure (b)). The plots show measured bandwidth at each peer along with the average value, varying the size of the message sent by each peer.

Figures 7 and 8 show the bandwidth measured by each peer when performing, respectively, a scatter and a gather as the size of the sent messages vary. Figure 7.a shows the results of a scatter performed from a slow sender to the others, while Figure 7.b depicts the scatter from a fast sender. Both the graphs exhibit the same trend. The gap in the graphs between the sender and the receivers are justified since the time to complete the experiment is of the same order of magnitude for all the receivers involved, but the sender must send much more data than the data received by the receivers. On the other hand, the bandwidths for scattering 1MB of data are all comparable. This is due to the fact that the packet size is too short to create congestion in the network. Hence, any direct measure of bandwidth must require at least 10MB of data. The same conclusion can be reached by looking at the unidirectional bandwidth experiments, noting that the bandwidth saturates when the messages reach a size of 10MB. Clearly these observations are only valid in the context of our toy experiments, where the underlying interconnection is a homogeneous LAN. Different thresholds are to be expected when performing the experiments on a wide area network.

Gather measurements involve similar considerations as scatter. What can be noted in Figures 8.a and 8.b is that the incoming bandwidth of the *slow* computer gathering data is larger than the outgoing bandwidth of the *slow* computer scattering data. On the converse, the incoming bandwidth of the *fast* computer gathering data, is larger than the outgoing bandwidth of the *fast* computer, when scattering data. This intriguing phenomenon needs extra investigation that will be carried out in our future work.

Finally, Figure 9 shows the results of CPU performance measurements. As explained before, the computation performed by the inquired peer is a simple random number generation, whose simple iteration is given by `seed=MOD(8121*seed+28411, 134456)`. A peer asks another peer to perform a number of iterations of the random number generator and

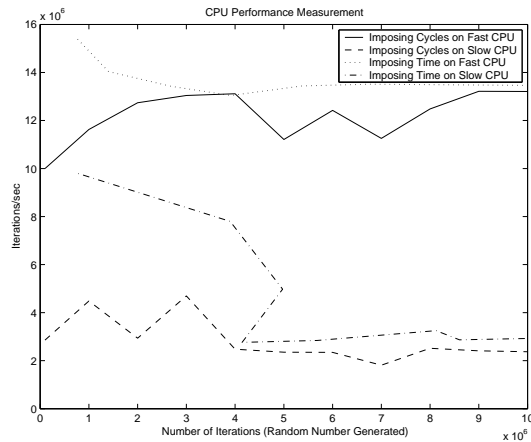


Figure 9: CPU power measurements of a fast and a slow computer. The measures has been executed by both imposing the number of iterations and the execution time of a simple random number generator.

then checks the correctness of the outcome. The plots shows the outcome of two tests: the first test sets the number of iterations to be performed by a remote peer and then measures the time, the second test sets the computing time and then evaluates the actual number of iterations performed. Even though the measurement is influenced by the computational load of the inquired peer, the results indicate that such a test may be employed if a sufficiently large number of iterations are executed by the remote peer to deal with the clock resolution.

References

- [1] N.M. Amato, J. Perdue, A. Pietracaprina, G. Pucci, and M. Mathis. Predicting performance on SMP's. A case study: The SGI Power Challenge. In *Proc. IEEE International Parallel and Distributed Processing Symposium, IPDPS2000*, 729-737, 2000.
- [2] G. Antoniu, P. Hatcher, M. Jan, and D. A. Noblet. Performance Evaluation of JXTA Communication Layers. In *Proc. Workshop on Global and Peer-to-Peer Computing (GP2PC 2005)*.
- [3] H. Balakrishnan, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica. Looking up data in P2P systems. In *Communications of the ACM*, February 2003.
- [4] A. Bertoldo, M. Bianco, and G. Pucci. A Static Parallel Multifrontal Solver for Finite Element Meshes. University of Padova, Italy. Available at <http://www.dei.unipd.it/~cyberto/tr-04-01.pdf>.

- [5] M. Bianco and G. Pucci. On the predictive quality of BSP-like cost functions for NOWs. In *Proc. of Euro-Par2000 Parallel Processing*, pages 638–646. Lecture Notes in Computer Science 1900, Springer-Verlang, 2000.
- [6] R.L. Carter, M.E. Crovella. Measuring Bottleneck Link Speed in Packet-Switched Networks. Technical Report BU-CS-96-006, Boston University, Computer Science Department, 1996.
- [7] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, S. Shenker. Making Gnutella-like P2P Systems Scalable. In *SIGCOMM'03*, 407–418, Aug. 2003.
- [8] F. Dabek, R. Cox, F. Kaashoek, R. Morris. Vivaldi: A decentralized Network Coordinate System. *SIGCOMM'04*, 15–26, 2004.
- [9] F. Dabek, J. Li, E. Sit, J. Robertson, F. Kaashoek, R. Morris. Designing a DHT for Low Latency and High Throughput. *SIGCOMM'04*, 15–26, 2004.
- [10] N. Drost, R. Nieuwpoort, H. E. Bal. Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing. In *Proc. IEEE Int. Symp. on Cluster Computing and the Grid, (CCGRID'06)*, 2006
- [11] R.J. Dunn, J. Zahorjan, S.D. Gribble, H.M. Levy. Presence-Based Availability and P2P Systems. *Peer-to-Peer Computing 2005*. 209-216, 2005
- [12] G. Fedak, C. Germaine, V. Nèri, F. Cappello XtremWeb: A Generic Global Computing System. In *Proc. IEEE Int. Symp. on Cluster Computing and the Grid, (CCGRID'01)*, 2001.
- [13] P. Francis, S. Jamin, C. Jin, D. Raz, Y. Shavitt, L. Zhang. IDMaps: A Global Internet Host Distance Estimation Service *IEEE/ACM Transactions on Networking*, (9)5:525–540, October, 2001.
- [14] S. Genaud, C. Rattanapoka. A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs. In *Proc. EuroPVM/MPI 2005*, 276–284, 2005.
- [15] J.D. Gradecki *Mastering JXTA: Building Java Peer-to-Peer Applications*. Wiley, 2002.
- [16] GIMPS: Great Internet Mersenne Prime Search. www.mersenne.org
- [17] W. Gropp, E.L. Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting*, LNCS 1697, 11–18, 1999
- [18] D. Grove, P. Coddington. Precise MPI performance measurement using MPIBench. In *Proceedings of HPC Asia*, September 2001.

- [19] K.P. Gummadi, R.J. Dunn, S. Saroiu, S.D. Gribble, H.M. Levy, J. Zahorjan. Measurements, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOPS'93)*, 314–329, 2003.
- [20] K.P. Gummadi, S. Saroiu, S.D. Gribble. King: Estimating Latency Between Arbitrary Internet End Hosts. In *Proc. 2nd SIGCOMM Internet Measurements Workshop (IMW 2002)*, 2002.
- [21] N. Hu, L. Li, Z.M. Mao, P. Steenkiste, J. Wang. Locating Internet Bottlenecks: Algorithms, Measurements, and Implications. In *Proc. ACM SIGCOMM'04*, 41–54, 2004.
- [22] N. Hu, P. Steenkiste. Evaluation and Characterization of Available Bandwidth Probing Techniques. In *IEEE J. on Selected Areas in Communications*, **21**(6), 879–894, 2003
- [23] N. Hu, P. Steenkiste. Locating Internet Bottlenecks: Algorithms, Measurements, and Implications. In *SIGCOMM'04*, 41–54, 2004 187-192, 2005
- [24] N. Hu, P. Steenkiste. Exploiting Internet Route Sharing for Large Scale Available Bandwidth Estimation. In *Proc. IMC'05, 2005 Internet Measurement Conference*, 187-192, 2005
- [25] N. Hu, P. Steenkiste. Emodis - An End-Based Network Monitoring and Diagnosis System. Technical Report CMU-CS-05-146, Carnegie Mellon University, 2005., 187-192, 2005
- [26] N. Hu, O. Spatscheck, J. Wang, and P. Steenkiste. Optimizing network performance in replicated hosting. In *The Tenth International Workshop on Web Caching and Content Distribution (WCW 2005)*, September 2005.
- [27] V. Jacobson Pathchar. <ftp://ftp.ee.lbl.gov/pathchar/>, 1997.
- [28] “The JXTA bench project”. bench.jxta.org.
- [29] E. Kalynianaki, I. Pratt. Building Adaptive Peer-to-Peer Systems. In *Proc. The Fourth IEEE Int. Conf. on Peer-To-Peer Computing*, 268–269, Aug. 2004.
- [30] M. Kleis, X. Zhou. A Placement Scheme for Peer-to-Peer Networks Based on Principles from Geometry. In *Proc. The Fourth IEEE Int. Conf. on Peer-To-Peer Computing*, 134–141, Aug. 2004.
- [31] K. Lakshminarayanan, V.N. Padmanabhan. Network Performance of Broadband Hosts. Microsoft Research, Technical Report MSR-TR-2003-15, 2003.
- [32] K. Lai, M. Baker. Nettimer: A Tool for Measuring Bottleneck Link Bandwidth. In *Proc. 3rd USENIX Symposium on Internet Technologies and Systems*, 2001.

- [33] T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.
- [34] V. Lo, D. Zhou, D. Zappala, Y. Liu, and S. Zhao. *Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet*. In The 3rd International Workshop on Peer-to-Peer Systems IPTPS'04.
- [35] E. Ng, H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approach. In *Proc. IEEE INFOCOM 2002*, 2002.
- [36] T. Qiu, F. Wu, G. Chen. A Generic Approach to Make Structured Peer-to-Peer System Topology-Aware. In *Proc. Third International Symposium on Parallel and Distributed Processing and Applications (ISPA '2005)*, 816–826, 2005.
- [37] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM'01*, 2001.
- [38] S. Ratnasamy, M. Handley, R. Karp, S. Shenker. Topologically Aware Overlay Construction and Server Selection. In *Proc. of IEEE INFOCOM'02*, 2002.
- [39] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz. Handling Churn in a DHT. In *Proc. USENIX Annual Technical Conference*, 127–140, Boston, MA, USA, June 2004.
- [40] A. Rowstron, P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large Scale Peer-to-Peer Systems. In *Proc. IFIP/ACM Int. Conf. on Distributed System Platforms (Middleware)*, 2001.
- [41] R.H. Saavedra, R.S. Gaines, M.J. Carlton. Characterizing the Performance Space of Shared Memory Computers Using Micro-Benchmarking. Technical Report USC-DC-93-547, University of Southern California, 1993.
- [42] S. Saroiu, P.K. Gummadi, S.D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. of Multimedia Computing and Networking 2002 (MMCN'02)*, 2002.
- [43] S. Saroiu, P.K. Gummadi, S.D. Gribble. SProbe: A Fast Technique for Measuring Bottleneck Bandwidth in Uncooperative Environments. In *Proc. IEEE INFOCOM 2002*, 2002.
- [44] S. Savage. Sting: a TCP-based Network Measurement Tool. In *Proc. 1999 USENIX Symp. on Internet Technologies and Systems*, 71–79, 1999.
- [45] SETI@Home Project. setiathome.berkeley.edu
- [46] M. Srivatsa, B. Gedik, L. Liu. Scaling Unstructured Peer-to-Peer Networks With Multi-Tier Capacity-Aware Overlay Topologies. In *Proceedings Tenth International Conference on Parallel and Distributed Systems (ICPADS'2004)*, 17–24, July 2004.

- [47] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnam. Chord: a Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM'01*, 2001.
- [48] D. Thain, T. Tannenbaum, M Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4): 323-356, 2005.
- [49] G. Tsouloupas, M D. Dikaiakos. GridBench: A Workbench for Grid Benchmarking. In *Advances in Grid Computing - EGC 2005*, 211-225, 2005
- [50] L.G. Valiant. *A bridging model of parallel computation*. *Communication of ACM*, 33, 103–111, 1990.
- [51] K. Yotov, K. Pingali, P. Stodghill X-Ray: A Tool for Automatic Measurement of Hardware Parameters In *International Conference on Quantitative Evaluation of SysTems (QEST)*, 2005.
- [52] K. Yotov, K. Pingali, P. Stodghill Think Globally, Search Locally. In *International Conference on Supercomputing (ICS)*, 2005.
- [53] B.Y. Zhao, A.D. Joseph, J. Kubiawicz. Locality Aware Mechanisms for Large-scale Networks In *Proc. Workshop on Future Directions in Distributed Computing*, June 2002.

A JXTA overview

In this appendix we briefly describe the main characteristics of the JXTA environment used in designing our preliminary set of experiments.

Project JXTA (subsequently referred to as simply JXTA) [15] is a collection of six protocols intended to form a P2P network. These protocols provide functionalities and/or interfaces to allow a developer to build an application based on a P2P network. So, the developer can use his preferred network topology disregarding the real network topology and policies (firewall, gateway, NAT, etc.). Moreover the network topology can be changed dynamically during the execution of the execution.

The JXTA architecture consists of three layers. The lower layer is the *core* level, which deals with the communication among the peers. It can choose among different known protocols to exchange messages between two or more peers. At present, JXTA can use TCP/IP, Tls, Beep, HTTP, and ServletHTTP. By employing these protocols, the core creates an overlay network over the existing network facilities. The *service* layer, built on top of the core layer, provides higher-level functionalities needed by the top layer, called the *application* layer, where the applications are ultimately created and executed.

Each layer has access only to the services provided by the layer directly below it, while it completely ignores the implementation details of such a layer. So the developer

is allowed to ignore the details of the implementation of a layer below those he needs to work on.

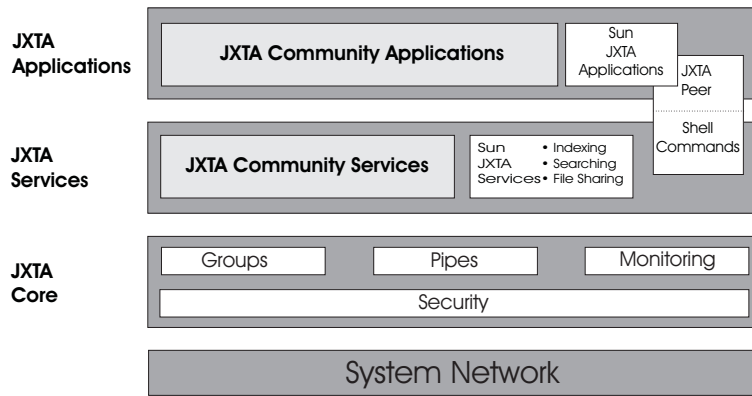


Figure 10: Project JXTA layer.

JXTA technology is designed to provide users with:

- *Interoperability*, to enable interconnected peers to easily locate and communicate with each other, participate in community-based activities, and offer services seamlessly across different P2P systems and different communities.
- *Platform independence*, to be independent on programming languages (e.g., C or Java), system platforms (such as the Microsoft Windows and UNIX operating systems), and networking platforms (e.g., TCP/IP or Bluetooth).
- *Ubiquity*: to supply implementations on every device with a digital heartbeat, including sensors, consumer electronics, PDAs, network routers, desktop computers, data-center servers, and storage systems.

JXTA features six main elements, that are listed below.

ID. Every resource in the system is addressed by associating to it a *JXTA ID*, also referred to as a *URN* (Uniform Resource Name), which is a unique string used for the identification of six types of resources:

- Peers
- Peer groups
- Pipes
- Content
- Module classes
- Module specifications

Each ID is made of a type descriptor (peer, pipe, etc), and by a 128 bit string.

Advertisement. An advertisement is an XML document that describes a JXTA pipe, peer, peer group, or service. Advertisements follow standards for encoding tags and content. An advertisement is used to exchange information about what is available in the JXTA network.

Peer. A peer is an application running on a computer device, that has the ability to communicate with other peers. Based on specifications, a peer implements the protocols at core level on JXTA. A single *networked device* can have any number of JXTA peers running on it.

There are three kinds of peers:

- **Endpoint peer:** Endpoint peers are the standard application peers in the network.
- **Relay peer:** Relay peers (or *Relay super-peers*) bridge peers that do not have direct physical connectivity (e.g., if they are connected through Nats, firewalls, etc.), by providing the ability to spool messages for unreachable or temporarily unavailable edge peers. Relay peers play the role of landmarks, facilitating the routing of messages between not directly reachable peers.
- **Rendezvous peer:** Rendezvous are peers that agreed to cache advertisement indices (i.e. pointers to edge peers that cache the corresponding advertisement). Rendezvous peers conceptually corresponds to well known locations used for indexing and locating advertisements.

Endpoint peers maintain special relationship with their rendezvous peers. Rendezvous peers just maintain an index of advertisements published by their endpoint peers. By not caching advertisements, the rendezvous architecture results to be more scalable and avoids the problem of caching out-of-date advertisements. The Shared-Resource Distributed Index (SRDI) service is used by endpoint peers to index their advertisements on rendezvous peers and to push advertisement indices when publishing new advertisements. Indices can be pushed synchronously when a new advertisement is published, or asynchronously by a SRDI daemon that polls at fixed intervals. Finally, endpoint peers use a DHT to query the right Rendezvous peer to obtain information about the searched advertisement.

Peer Group. Peers in the JXTA network self-organize into peer groups. A peer group represents a dynamic set of peers that have agreed upon a common set of policies (membership, content exchange, etc.). Each peer group is uniquely identified by a unique peer group ID. JXTA does not dictate when, where, or why peer groups are created. JXTA only describes how a peer group is created, published, and discovered. At boot time, every peer joins the NetPeerGroup. The NetPeerGroup acts as a root peer group which every peer initially belongs to. Users, service developers, and network administrators can dynamically create peer groups for scoping interaction between peers, and for matching their applications demands. JXTA recognizes three main motivations for creating peer groups:

- *To create secure domains for exchanging secure contents.* Peergroups virtualize the notion of routers and firewalls, subdividing the network into secure regions without respect to actual physical network boundaries.
- *To create a scoping environment.* Peergroups partition the network into abstract regions, providing an implicit scoping mechanism for restricting the propagation of discovery and search requests.
- *To create a monitoring environment.* Peergroups allow monitoring (traffic inspection, accounting, tracing) of peers for any purpose.

Pipe. Pipes are virtual communication channels used to send and receive messages between services and applications (peers). Pipes provide a virtual abstraction over the peer end-points to provide the illusion of virtual incoming and outgoing mailboxes that are not physically bound to the specific peer location. Pipes can connect to one or more peer end-points. At each endpoint a program is assumed to deal with sending, receiving, or managing message queues or streams. Pipe end-points are referred to as *the input pipe* if it is at the receiving side, or *the output pipe* if it is at sending side.

Pipe end-points are dynamically bound to involved peers at runtime. Using the pipe abstraction, applications and services can transparently failover from one physical peer endpoint to another in order to mask a service or peer failure, or to access a newly published instance of a service. When a message is sent into a pipe, the message is sent by the local output pipe to the destination input pipe currently listening to this pipe.

Pipes offer two modes of communication:

- *A point-to-point pipe* that connects exactly two pipe end-points with a **unidirectional and asynchronous channel**. No reply or acknowledgment operation is supported. The message payload may also contain a pipe advertisement that can be used to open a new pipe to reply to the sender (send/response).
- *A propagate pipe* that connects one output pipe to multiple input pipes. Messages flow from the output pipe end (propagation source) into the input pipe ends. The propagate message is sent to all listening input pipe ends in the current peergroup context. This process may create multiple copies of the message. The implementation may involve the use of low-level multicasts (IP), if available, or be based on point-to-point protocols (e.g., HTTP).

The messages that can be exchanged in JXTA are of two different kinds. The first type are packets containing XML data. The second type of messages may deliver arbitrary binary content (wrapped in an XML message).

Modules. Modules are the way in which the developers provide new services in a JXTA network. Any new service is described and made available by modules. Three kinds of modules are needed to provide a new service in JXTA, namely:

- *Module class* that describes the functionality of the service.
- *Module specification*, that is a description of a class. It provides information on the invocation of the service (for example, a pipe advertisement).
- *Module implementation* that is an implementation of the specification.

Protocols. Below is a brief overview of the six JXTA protocols with the indication of the architectural level they belong to.

- **Endpoint Routing Protocol (ERP)** [core layer]: defines a set of request/query messages that are processed by a routing service to help a peer route messages to their destination.
- **Peer Resolver Protocol (PRP)** [core layer]: provides a generic query/response interface that applications and services can use for building resolution services. The PRP provides the ability to issue queries within a peer group and later identify matching responses.
- **Rendezvous Protocol (RVP)** [service layer]: The Rendezvous Protocol (RVP) is used for propagating messages within a peer group. The Rendezvous Protocol provides mechanisms which enable propagation of messages to be performed in a controlled way. To improve efficiency in message propagation, some peers may agree to do extra work, i.e., each Rendezvous Peer cooperates with other Rendezvous Peers and with client peers to propagate messages amongst the peers of a peer group. The Rendezvous Peers cooperate to form a PeerView. The PeerView is a list of the peers which are currently acting as Rendezvous Peers. The PeerView is structured such that Rendezvous Peers are able to direct messages within the peer group in a consistent way without any centralized coordination.
- **Peer Discovery Protocol (PDP)** [service layer]: is used to discover any published peer resource. Resources are represented as advertisements. Every resource, be it a peer, a peergroup, a pipe, a module, etc., must be represented by an advertisement.
- **Peer Information Protocol (PIP)** [service layer]: provides a set of messages to obtain peer status information. PIP is an optional JXTA protocol. Peers are not required to respond to PIP requests.
- **Pipe Binding Protocol (PBP)** [service layer]: is used by applications and services in order to communicate with other peers. PBP is layered upon the *Endpoint Protocol*, and may use a variety of Message Transports such as the JXTA HTTP Transport, the JXTA TCP/IP Transport, or the secure JXTA TLS Transport to send messages. Actual pipe implementations may differ, but all the compliant implementations must use PBP to bind the pipe to an endpoint.

Class name	Description
JXTAMulticastSocket	The JXTAMulticastSocket class is useful for sending and receiving JXTA multicast packets.
JXTAServerSocket	JXTAServerSocket is a bi-directional pipe that behaves very much like ServerSocket.
JXTASocket	JXTASocket is a bi-directional pipe that behaves very much like a socket: it creates an InputPipe and listens for pipe connection requests.
JXTASocketAddress	This class implements a JXTASocket address (PeerGroup ID + Pipe Advertisement + (optional) Peer ID).
JXTASocketOutputStream	This class implements a buffered output stream.

Table 1: A brief description of the JXTASocket interface

JXTA APIs. Project JXTA provides some useful APIs to ease the writing of applications, one of the most useful being the *JXTASocket package*. JXTASocket provides a socket-like interface over JXTA pipes. Since the PipeService provides uni-directional and unreliable communication channels, JXTASocket employs its own protocol over pipes to establish bi-directional connections for every request. In addition a JXTASocket also employs the *reliability library* to ensure reliable data exchange. Another feature of JXTASockets is chunking, which relieves applications from having to worry about MTU sizes. The JXTASocket interface is briefly described in Table 1. To create a socket's communication channel a user needs to provide the following parameters to JXTA: PeerGroup, PipeAdv, and PipeId. These parameters can be directly passed to the socket class constructor or used to build a JXTASocketAddress object. A communication can be instantiated by passing the JXTASocketAddress object to the socket. The third parameter, PipeId is optional.

Many features can be handled using JXTASocket. The more important ones are:

- TimeOut: This sets the timeout to send and receive data and the waiting time before applying any change to the socket itself (e.g. close);
- BackLog: the maximum length of the message queue;
- isStream: a boolean flag used to toggle the reliability of the channel.

Another possible high-level interface for data exchange is provided by *BiDiPipe*. JXTABiDiPipe is a bi-directional Pipe, it creates an InputPipe for incoming Messages, and a messenger for outgoing messages. JXTABiDiPipe defines its own protocol for negotiating connections.