Towards Self-Managing QoS-Enabled Peer-to-Peer Systems

Vana Kalogeraki, Fang Chen, Thomas Repantis, Demetris Zeinalipour-Yazti

Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521
E-mail: {vana, fchen, trep, csyiazti}@cs.ucr.edu

Abstract. Peer-to-peer systems that dynamically interact, collaborate and share resources are increasingly being deployed in wide-area environments. The inherent ad-hoc nature of these systems makes it difficult to meet the Quality of Service (QoS) requirements of the distributed applications, thus having a direct impact on their scalability, efficiency and performance. In this paper we propose adaptive algorithms to meet applications QoS demands and balance the load across multiple peers. These comprise (a) resource management mechanisms to monitor resource loads and application latencies and (b) self-organization algorithms to dynamically select peers that maximize the probability of meeting the applications' soft real-time and QoS requirements. Our algorithms use only local knowledge and therefore scale well with respect to the size of the network and the number of executing applications.

1 Introduction

In the last few years, the new emerging Peer-to-Peer (P2P) model has become very attractive for developing large scale file systems [1–6] and sharing resources (i.e., CPU cycles, memory, storage space, network bandwidth) [7, 8] over large scale geographical areas. This is achieved by constructing an overlay network of many nodes (peers) built on top of heterogeneous operating systems and networks. P2P systems present the evolution of the client-server model that was primarily used to manage small-scale distributed environments. The most distinct characteristic in the P2P overlays is that there is symmetric communication between the peers; each peer has both client and server role.

Many efforts have been made to improve resource usage, minimize network latencies and reduce the volume of unnecessary traffic incurred in large-scale P2P overlays. Two main approaches have emerged for constructing overlay networks: *Structured* and *Unstructured* overlays. *Structured* overlay networks [4, 3, 5] are organized in such a way that objects are located at specific nodes in the network and nodes maintain some state information, to enable efficient retrieval of the objects. These sacrifice atomicity by mapping objects to particular nodes and assume that all nodes are equal in terms of resources, which can lead to bottlenecks and hot-spots. In *unstructured* overlay networks, on the other hand,

objects can be located at random nodes, and nodes are able to join the system at random times and depart without a priori notification. Recent efforts have shown that a self-organizing unstructured overlay protocol maintains an efficient and connected topology when the underlying network fails, performance changes, or nodes join and leave the network dynamically [9]. More advantages of unstructured overlay networks include their ability for self-organization, for adaptation to different loads, and for resiliency to node failures. Several efforts have demonstrated that P2P systems can be used efficiently in the context of multicast [10], distributed object-location [4, 3] and information retrieval [11].

However, hosting distributed, real-time applications with Quality of Service (QoS) demands, such as predictable jitter and latency on P2P systems imposes many challenges. These types of applications have distinctly different characteristics from content-based or multicast applications traditional being deployed on P2P systems. Examples of such applications include industrial process control systems, avionics mission computing systems and mission-critical video processing systems [12].

For example, consider a surveillance system that transfers public health, laboratory, and clinical data over the Internet. In this example, both continuous and discrete data (such as text, images, audio and video streams and control information) needs to be collected from multiple nodes in the system. Personnel will then analyze the gathered data quickly and accurately to monitor disease trends, identify emerging infectious diseases or track potential bioterrorism attacks. These have end-to-end soft real-time and QoS requirements on data transmission, including fast and reliable transfer, and substantial throughput. In addition, the audio and video streams may need to be transcoded to different formats or presentations (such as lower resolution) to transmit the data over resource constrained links. To support the QoS demands of the distributed applications, the P2P system must be flexible, predictable and adaptable.

Distributed and real-time applications have been successfully developed over middleware technologies, such as OMG's Common Object Request Broker Architecture (CORBA)[13], Microsoft's Distributed Component Object Model (DCOM) [14]), Sun's Java Remote Method Invocation (RMI) [15] and the Simple Object Access Protocol (SOAP) [16]. These typically rely on local management or the use of centralized managers that have a global view of the system [17], [18], [19], [20], [21].

In our view, the inherent advantages of the P2P systems, including scalability, decentralization and ease of use makes it feasible to develop large-scale distributed and real-time applications. However, current P2P systems are limited in capability because of lack of automated and decentralized management mechanisms. There are two main reasons for this limitation: (1) in a large scale system, each node cannot have an accurate global view of the system at all times, since the state of the system changes much faster than it can be communicated to the peers, and (2) the P2P infrastructure can encompass resources with different processing and communication capabilities, therefore, distributed applications that execute over wide-area environments are subject to greater

variations due to unpredictable communication latencies and changing resource availability.

QoS properties in the P2P systems can be enabled in two ways: *statically*, where we must ensure that adequate resources are available before the application execution, or *dynamically*, where the resource usage is adjusted based on runtime system monitoring. Examples of static QoS properties include peer geographic location or specific platforms and hardware resources available at the peers. Examples of dynamic QoS properties include runtime resource reallocation and re-prioritization to handle resource failures or changes in the CPU and network load.

The objective of this work is to build self-managing large-scale P2P systems that are able to meet application QoS requirements. To achieve this, we propose to use self-organization algorithms that revise peer connections dynamically to minimize application latencies and distribute the resource load. These work together with local resource management mechanisms for managing CPU and network bandwidth and prioritizing application requests, and system-wide management mechanisms that run across multiple peers to improve task execution latencies.

Towards this view we present an architecture with two important components:

- A resource management framework to meet the end-to-end soft real-time and QoS requirements of the distributed applications. The framework consists of mechanisms for managing the local resources, prioritizing application requests and propagating resource and timing measurements system-wide. These mechanisms are decentralized, adaptive and use only local information.
- Adaptive self-organization algorithms that improve application latencies and balance the load across multiple peers to meet their end-to-end soft real-time and QoS requirements. The decisions are made in a decentralized manner, thus achieving system scalability.

We implemented the self-organization algorithms in our P2P middleware that uses an unstructured communication protocol to establish connections between the peers. We present empirical results over our P2P system that demonstrate the adaptability, predictability and performance of our resource management mechanisms.

The rest of the paper is organized as follows. Section 2 gives an overview of our P2P architecture and presents the system model and metrics. In Section 3 we describe our self-organization algorithms. In Section 4 we discuss the experimental results. Section 5 presents related work and Section 6 concludes the paper.

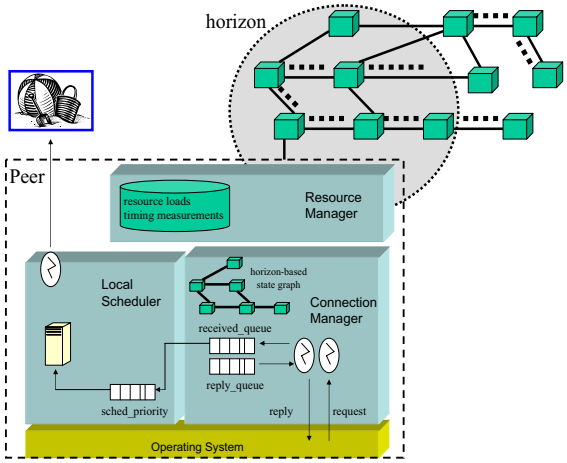

Fig. 1. Our P2P System Architecture.

2 Design and Implementation Overview

A P2P system is modeled as an overlay network of nodes (peers[1]) in which each peer comprises a Connection Manager, a Resource Manager and a Local Scheduler, as shown in Figure 1.

The Connection Manager is responsible to manage the peer connections. Each node keeps a small number of connections to other peers; the number of connections is typically limited by the network bandwidth at the peer. Typically, a XP 2000 Athlon PC workstation behind a 100Mbit/s network connection can easily support 20 concurrent peer connections, while the same machine behind a modem 56kbit/s connection can support 1-2 connections. As a result, remote peer invocations may take a long time to complete due to nodes with high latencies or limited network bandwidth, affecting the end-to-end performance of the distributed applications. Peer connections are established as a result of new peers joining the system or are triggered by the self-organization algorithms in which each node tries to connect to better peers. For each peer connection q, the Connection Manager at node p maintains the $Geographical\_info_q$ that consists of the network address and port number of the peer {IP address, port number} and $Peer\_type_q$: whether the peer is immediate or indirect. The Connection Manager creates and manages a number of connection threads for each peer connection used to handle request and response messages coming from that peer. All the requests coming from the peers, enter the Connection Manager's receiving queue.

[1] Two nodes p and q are called `immediate` peers if there is a direct connection between the nodes. Two nodes p and q are called `indirect` peers if there exists a communication path between the nodes. In some cases, two peers may not be connected at all.

The Connection Manager constructs and maintains a horizon-based state graph that stores peer resource utilization and application timing measurements and also captures the relationships between the peers. The Connection Manager obtains this information either by propagating updates periodically between the peers, or by recording the parameters carried along with the messages. Note, that, because of the large scale and dynamic nature of the system, the state graph is local at each node and captures only a partial (limited) view of the system. This view is bound by the horizon of the peer. The state graph is constructed and updated dynamically based on the applications executing in the system and the connections established and torn down by the peers.

The Connection Manager works in concert with the Local Resource Manager that controls and monitors the access to the node's local resources (*e.g.,* CPU, memory and network bandwidth) and profiles the behavior of the applications as they execute. For example, if the Resource Manager reports that the processor is overloaded (receiving queue size is full), the Connection Manager does not accept new requests. These requests will be propagated to the node's peers. Our previous measurements [22] indicate that such profiling can be done at run-time with less than 1% overhead, by invoking the `/proc` interface, but (1) a history of measurements must be maintained and (2) the profiling frequency must be carefully tuned dynamically to adequately capture the load fluctuations in the peer.

The Local Scheduler in the node is responsible for specifying a local ordered list (schedule) for the application object invocations based on the scheduling algorithm implemented in the system. Our scheduling algorithm is based on the Least Laxity Scheduling (LLS) algorithm that allows us to capture timing delays as the applications execute across multiple processors in the system [23, 24].

2.1 Application Tasks

Users request applications from the system that trigger the execution of tasks. An application task is defined as a sequence of invocations of objects distributed across multiple peers in the system. The execution of the task starts at the user invocation and completes when a result is returned back to the user. A task is executed by a single thread executing in sequence on one or more peers. The execution times of the tasks are affected by the load on the peers and the latencies on the communication links. If a node cannot execute the task locally, it propagates the request to one of its peers. This process continues until an appropriate node is found to execute the request. To provide a termination condition so that requests are not propagated indefinitely in the network, we associate a time_to_live (ttl) value with each task that determines the maximum number of hops the task will propagate in the system. To avoid loops in task propagation, we choose not to propagate requests that have previously arrived at the same peer. When the execution finishes, a reply is generated that follows the same path to be reported back to the user. Users may trigger the execution of multiple tasks concurrently and asynchronously.

Each task t is characterized by the $Task\_id_t$ which is a unique identifier that distinguishes each task from the others, generated by the peer initiator of the task. We represent as $Task\_type_t$ the type of task (to be executed), carried along with the message. $Deadline_t$ is the time interval, starting at user invocation within which the task t must complete, specified by the user. $Project\_latency_t$ is the estimated amount of time required for the task to complete. This includes queueing delays and the latencies on communication links. $Laxity_t$ is computed as the difference between the deadline and the projected latency of the task. The laxity value determines the order with which the task will be executed in the system. The task with the smallest laxity value has the highest priority.

3 System Resource Management

In this section, we describe our self-organization algorithm that uses resource and timing measurements monitored locally and collected from remote peers.

3.1 Resource and Timing Measurements

The Resource Managers at the processors monitor the execution of the tasks across multiple peers and record the peer-to-peer messages exchanged. The P2P communication protocol [2] enables interoperability across peers on different nodes in a large scale system and implemented using different languages. The Connection Managers communicate through five message types (`ping, pong, request, reply, update`), of which the ping and pong messages are used to establish connections with remote peers. The Connection Manager sends a `ping` message to establish a connection with a peer. A peer that receives the ping message and chooses to accept the connection, replies with a `pong` message that includes its IP address and port number for the requesting node to connect to. The decisions to which peers to connect to or whether to accept an incoming connection, are made by our self-organization algorithm.

For a remote task invocation, the Connection Manager constructs a `request` message that carries the task operation. A request message includes the identifier of the task `task_id`, a `descriptor_id` that uniquely characterizes the peer that propagated the task last, and a `hop_count` that determines the maximum number of times the task will be propagated to the system before it expires. When the task finishes execution, the Connection Manager will generate a `reply` message that carries along the return value of the invocation. It uses the descriptor_id carried along with the messages, to propagate the result, through the same path, back to the user. The Resource Manager attaches a timestamp with each of the messages to measure peer connection times, local computation times and remote task execution times.

The Resource Manager measures the local execution time of the tasks that includes the processing time of the task at a peer and the queueing time at the local Scheduler's task queue. The processing time of the task depends on the type of object to be executed, the parameters carried along with the task

and the speed of the processor. The queueing time is affected by the priority (laxity value) of the task and the number of tasks currently being waiting at the Scheduler's queue.

Upon the receipt of a reply message, the Resource Manager measures the end-to-end latency of the task, as the time required for the task to complete, starting at user invocation until the reply message is received back at the user. Thus, the projected latency of the task includes the transmission times to propagate the task from one peer to another and the local execution time of the task. The transmission times are affected by the number of hops the task is propagated and the available bandwidth on the communication links. The Resource Manager measures the percentage of the processing load and the amount of memory used during the task execution. It obtains this information by using system calls to the `/proc` interface.

3.2 Resource and Timing Measurement Propagation

Self-organization is greatly affected by the frequency with which tasks execute in the system and resource and timing measurements are propagated to the peers. The Connection Manager triggers propagation of such measurements in two cases: (1) as a result of the Resource Manager feedback that measures new resource loads and timing measurements at the peer, and (2) when the Connection Manager detects an incoming or withdrawn peer connection. For example, when a connection with a new peer is established, the Connection Manager propagates its current resource and timing measurements to that peer.

Assuming that node p has m immediate peers, the Connection Manager at p constructs an `update` message that carries an array of length l for each of the m immediate peers. The array is constructed based on the measurements stored at p's horizon-based state graph which capture resource and timing information up to l hops away. By bounding l to a small number, we can control the amount of information propagated to the peers. Each entry in the array includes the following information: *(IP address, port num, CPU load, network bandwidth, immediate peers)*. Further, to regulate the rate of update propagation, the Connection Manager can choose to send an update only if the resource measurements have increased above an upper bound `HIGH` or if the peer is underutilized (below `LOW` bound). The advantage is that the amount of network traffic is minimized.

Upon the receipt of an update message from an immediate peer p, node q updates its local horizon-based state graph with p's most recent resource and timing measurements. The Connection Manager detects a new indirect peer, if the array has a new entry and thus updates its state graph. Similarly, if a peer has disconnected, the Connection Manager marks the peer as disconnected and updates the corresponding connection times in the graph.

There are two importance observations in the measurement propagation. First, there is a trade off between the accuracy of the resource utilization information maintained by the Connection Manager peers and the frequency of the update propagation. The higher the propagation frequency, the more accurate the measurements stored. However, a high propagation frequency incurs a high

penalty due to the large number of messages that have to be sent. Second, the accuracy of the information decreases as the number of hops between the peers increases. To remedy this, we introduce levels of confidence through weights $(w_0, w_2, ..., w_{l-1}), \sum_{i=0,..,l-1} w_i = 1$, where the higher confidence goes to peers one hop away.

3.3 Self-Organization

The goal in the peer-based organization algorithm is to improve task execution times by connecting to faster or less loaded peers.

The Connection Manager uses the resource load measurements collected at its horizon-based state graph, to estimate the projected latency of the tasks at the immediate and indirect peers. Let ρ_p be the average load on peer p and τ_{tp} be the mean processing time of task t on peer p. Assuming that ρ_c is the average load on the communication link c and σ_{tc} is the mean transmission time on each communication link c, the Connection Manager (using an M/M/1 queueing model) computes the projected latency of the tasks at peer p as:

$$Projected\_Latency_p = \sum_t \frac{\sigma_{tc}}{1-\rho_c} + \frac{\tau_{tp}}{1-\rho_p}$$

Once the projected task latencies have been estimated, the Connection Manager evaluates the relative benefit of its peers. It uses a utility function based on the resource loads and the task computation and communication latencies. Each node computes the utility of both its immediate and indirect peers and tries to connect to indirect peers with the highest utility. These are the peers that have the highest probability of meeting the soft real-time requirements of the tasks.

The Connection Manager at node p estimates the effects on the task latencies by considering the effects of increased or decreased processor loads and communication latencies on the times required to execute the tasks. The Connection Manager estimates the increased latencies of the tasks currently executed at p as a result of the new peer connection. A similar estimate is made for the reduced times of the tasks run in the vicinity.

Thus, the Utility value of both its immediate and indirect peers, as follows:

$$Peer\_Util_p(t) = \alpha * Peer\_Util_p(t-1) + \beta * e^{-Proj\_Latency_p}$$

where α and β are used to balance between new and previously computed utility values ($\alpha + \beta = 1$). By using exponentially weighted averaging it allows us to track current behavior with a large value yielding rapid response to changing conditions, and a small value yielding more smoothing and less noise. If the types of the executing tasks are stable, our algorithm approximates good peers accurately. If the behavior changes dynamically, the stability of the system is affected by the rate with which each node evaluates its peers and tries to connect to better ones.

The peer-based algorithm determines important indirect peers as peers with high utility values. It identifies immediate peers with low utility values as those

where the projected latency of the tasks propagated through those peers increases and the tasks start missing their deadlines. Thus, the Connection Manager at p identifies the peer q with the highest $Peer\_Util_q$ value for node p and peer s with the lowest $Peer\_Util_s$ as a candidate for replacement. Then, it probes peer q for a connection, by generating and sending a `ping` message. If the remote peer q accepts the connection, it replies with a `pong` message including its geographical information {IP address, port number} to allow peer p to connect to. If the maximum number of connections at p has exceeded, the Connection Manager chooses to disconnect from the least important immediate peer.

3.4 Dynamic System Operation

In a large-scale system, the availability of the resources changes as a result of new nodes becoming available, existing nodes failing or disconnecting, and new tasks executing in the system. For example, the execution of a new task increases the load on the processors and requires new projections of the task latencies, triggering self-organization. If the relative utility of an indirect peer increases over time, the node attempts to move closer to that peer and connect to it directly. If the connection is acceptable, it is actually performed. As the maximum number of connections allowed is exceeded or the latency on a peer is too long, then immediate peers with less utility are removed.

One issue in self-organization is how to tear down connections from peers. If a node disconnects from a peer whose tasks are currently being executed to remote peers, the return path of the tasks may be disconnected and the result cannot be propagated back to the source. To avoid this problem, we define a prior_disconnection period, during which two node temporarily remain connected until the results from currently executing tasks are propagated back to the source. During this time, the disconnected peer does not accept new tasks for propagation or execution. Although this will incur some additional overhead, it will allow all the tasks to complete.

The effectiveness of the system is affected by the frequency with which each peer executes the self-organization algorithm to find peers with better utility values. This can affect the stability of our system. To address this issue we choose to restrict the maximum number of times per time interval that a peer can make re-connections. This time interval is determined by considering the characteristics of the tasks in the system.

4 Implementation and Experimental Evaluation

4.1 Experimental Setup

To evaluate the working and performance of our self-organization algorithms we performed empirical experiments. The platform for our implementation consisted of Athlon XP2000 processors and Intel Pentium IV processors with 1GB memory, running Mandrake Linux 9.0, over a 100Mbit/s network. We built a

P2P system running on these machines. The peers are implemented using the C++ language and are multi-threaded. To simulate peers of different bandwidth capabilities we limited the sending and receiving speed of the peers. Figure 2 illustrates the initial topology of the system. The thin lines represent peers with slow communication links. The bandwidth on those slow peers was restricted to 200KBytes, the bandwidth on the remaining peers was set to 1MByte. Slow communication links introduce higher transmission overhead, especially when the source sends messages at a high transmission speed. In those cases, self-organization would be beneficial to both peers sending and receiving messages.

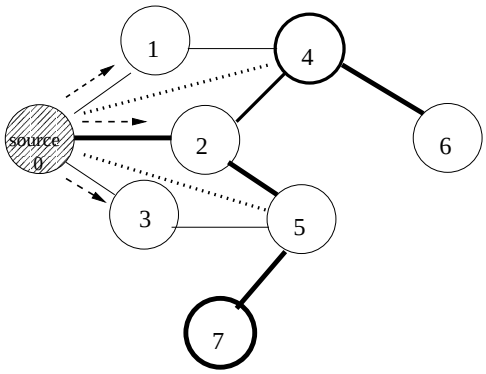

Fig. 2. System topology for the peer-based organization algorithm.

4.2 Application Tasks

We used soft real-time distributed multimedia tasks to drive the empirical evaluation of our system. In our scenarios, video streams were generated from the sources and transmitted over the network from one peer to the other until they reach the destination, where individual streams received from different sources are assembled and displayed separately. The multimedia tasks needed to be transcoded into a different format to reach resource-constrained client machines. Examples of transcoding operations include changing video compression formats, reducing video playback bit-rate and adjusting picture resolution.

The multimedia streams consist of a sequence of independent media units, in the case of MPEG-1 format these are called Group of Pictures (GOPs). For the experiments, our sources generated video streams of MPEG-1 format with a resolution of 320x240 and a variable bit rate (VBR) of about 900Kbps, each Group of Picture (GOP) consists of 12-13 frames which correspond to a 0.5 second playback time. The inter-arrival time between successive GOPs is 0.5 seconds. Transcoding services were implemented using the libavcodec library [25], which is an open source media library.

In the experiment, our transcoding task was to reduce the bit-rate from 900Kbit/s to 150Kbit/s. The request generator fetches a GOP, encapsulates it into a request message and forwards it to a peer. When the task finishes execution at a peer, the Connection Manager encapsulates the result in a reply message and sends it back to the request generator, following the same path. The tasks had to travel an average of 2 hops to get transcoded. The figure shows that the average transmission time is 70ms and the average transcoding time is computed to about 53 milliseconds. However, our experiments showed that the transmission times increase rapidly when network connections are slow (*e.g.,* 200Kbit/s) or when the tasks have to propagate more hops in the system. For GOPs of larger data size the transcoding time also increases, although the increase is small.

4.3 Performance Metrics

To evaluate the performance of our self-organization algorithms, we use the following metrics:

- *Miss ratio:* represents the percentage of tasks that miss their deadlines. The miss ratio is primarily affected by the utilization on the nodes and the communication links. As the load on the nodes or the transmission times of the tasks increase, the probability that the tasks miss their deadlines is higher.
- *Task execution time:* defined as the actual execution time of the task in the system. The execution time depends on the computation time of the task that includes transcoding time and queueing time, and the transmission latency experienced at each hop along the path it travels.
- *Estimated Projected Latency*: this is the estimated amount of time for the task to execute, projected by the Connection Manager at the peers. The task's projected latency depends on (1) the accuracy of the measurements recorded by the Resource Managers, (2) the feedback they provide to the Connection Manager and (3) the frequency with which Connection Managers propagate these measurements to their peers. For example, if the update frequency is low, the peers may not have recent resource information about their peers and therefore fail to estimate the task projected latencies accurately.

4.4 Self-organization algorithm

We conducted three experiments to measure the performance, accuracy and predictability of our self-organization algorithm.

End-to-End Task Execution Times. In the first experiment, we measured the average end-to-end task execution times as a function of the number of transcoding tasks executed in the system. The initial topology is shown in Figure 2, where the dotted lines represent the connections established as a result of the organization algorithm. In this experiment, each peer runs the peer-based

organization algorithm. Due to lack of space, we report results only for node 0. Note, that, node 0 is connected to one fast peer 2 and two slow peers $1, 3$. Transmission tasks are generated from node 0 with deadlines of 500ms. Resource and timing measurements are propagated at a rate of 200ms and self-organization is triggered every 3-5 seconds.

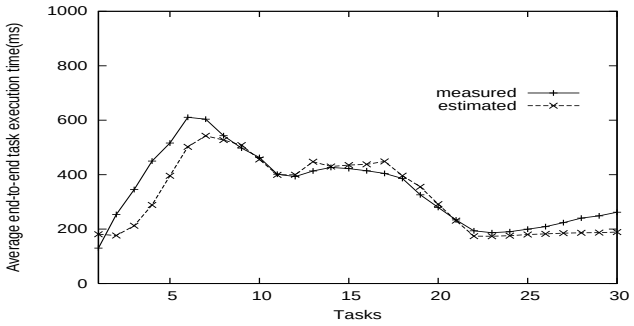

Fig. 3. Measured and estimated end-to-end task execution times as a function of the number of tasks being executed in the system.

Figure 3 shows the average end-to-end execution times of the tasks. The solid graph in the figure represents the actual execution times of the tasks, measured by the Resource Managers. The dotted graph in the figure represents the projected latencies of the tasks, estimated by the Connection Managers.

As tasks are generated, the execution times of the tasks increase. The reason is that because node 0 has two slow peers, the tasks are not accepted for execution at those peers, they are propagated to their own peers in the system. As a result, their transmission latencies increase. The Connection Manager at node 0 observes the increase in the task execution times, and triggers the peer-based organization algorithm. This will compute the relative utility values of the peers and will select the peer 4 with the highest utility to connect to (as shown with the dotted line in Figure 2). At this point, the maximum number of connections for the node has being exceeded, and the node chooses to disconnect from peer 1 that has low utility value. This reduces the end-to-end execution times of the tasks (shown by the decline in Figure 3). At a latter point during the execution, the Connection Manager discovers another peer 5 with a high utility value and chooses to connect to that peer directly and disconnect from its immediate peer 3. This will further improve the projected latencies of the tasks.

An important observation in this experiment is that the Connection Managers accurately estimates the projected latencies of the tasks at all times, even after organization. The reason is that the Connection Manager propagates the new resource measurements to their peers, which are stored in their horizon-based state graphs and are used to compute the new projected latencies for the tasks.

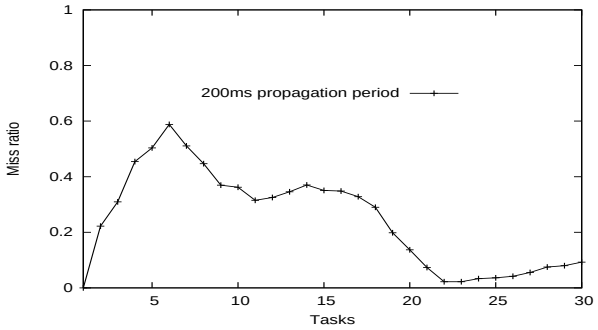

Fig. 4. Task miss ratio as a function of the number of tasks being executed in the system.

Miss Ratio. Figure 4 shows the corresponding improvement to the miss ratio of the tasks as a result of running the peer-based self-organization algorithm. The figure shows that when the execution times of the tasks increase, tasks start missing their deadlines. This is attributed to two factors: (1) the queueing delays in the local Schedulers' queues due to the large number of transcoding tasks concurrently being executed in the system, and (2) the transmission latencies experienced by the slow communication links and the number of hops that the tasks are being propagated. For example, when the execution times are 600ms, 60% of the tasks miss their deadlines. After the first organization, the task miss ratio drops to 35%. The second organization improves the task miss ratio even further and eventually very few tasks miss their deadlines.

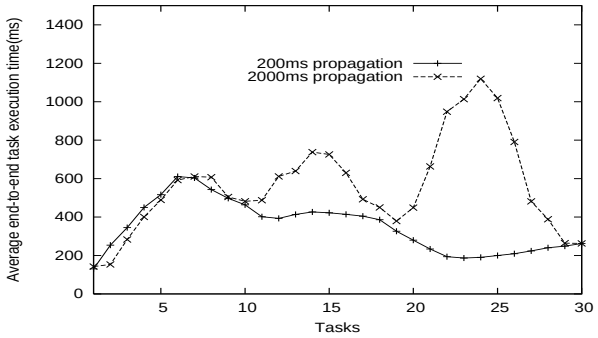

Fig. 5. End-to-end task execution times as a function of different propagation frequencies.

Effect of Propagation Frequency to Task Execution Times. In the last experiment of the peer-based organization algorithm, we evaluated the effectiveness of the resource and timing measurement propagation frequency to the task execution times (Figure 5). In these, we varied the frequency with which Connection Managers propagate feedback information to their peers from 200ms to 2000ms. Upon the receipt of new resource and timing measurements, the Connection Managers will use the new measurements to decide to which peers to

propagate tasks that cannot be executed locally, or whether they need to run the self-organization algorithm to connect to better peers.

When the frequency is high (200ms), the Connection Managers capture load fluctuations at their peers accurately, and therefore the queueing and transmission latencies of the tasks does not increase further. However, if the frequency is low, the Connection Managers do not accurately capture the load fluctuations at their peers. As a result, the end-to-end execution times of the tasks increase, and most of the tasks miss their deadlines. The frequency of propagation depends on the characteristics of the tasks (such as number and distribution of requests, and typical computation times) and resource and communication capabilities of the peers. Our experiments indicate that a frequency of 200ms is adequate to capture load fluctuations and transcode tasks end-to-end without missing their deadlines.

5 Related Work

The task of organizing a large network of peers for efficient data access is a very interesting problem that only recently has been addressed [26], [27], [28], [29], [30], [31]. However, the majority of this work has focused on file sharing applications.

The first wave of the P2P systems [1], [32], [2], [33], [34] perform poorly either because they rely on a centralized manager or they propose simplistic routing mechanisms. For example, Gnutella [2] relies on flooding the network with messages. Limewire [35] organizes the peers on static interest groups based on their preferred music category.

Distributed hash tables (DHTs) have been proposed as an alternative approach for organizing peer-to-peer systems [36], [4], [5], [3], [37], [38], [39] that improve performance by minimizing the number of hops to find the data. These consist of two components: (1) a consistent hashing over a one-dimensional space, and (2) an indexing mechanism to quickly navigate the space. These have the disadvantage that (a) assume that all peers are inherently equal in terms of resources, and (b) impose a a structure in the network by mapping objects to particular nodes and therefore may require a slow connection to be heavily utilized in order to discover a popular item. To the best of our knowledge, ours is the first work, that proposes self-organizing algorithms based on the dynamic properties of the peers to meet the distributed applications end-to-end QoS requirements.

Recent efforts recognize the need to improve the performance of the overlay network by partitioning peers into groups based on the round-trip time (RTT) of the messages. In these, peers in the same group are closes to each other in terms of latency. [40] *et al* present a binning scheme, and use landmark nodes to determine the relative latencies for the peer partitioning. In [41], the authors propose to construct an auxiliary network on top of the overlay network using BGP information, and choose neighboring peers based on some random landmark nodes. Eugene *et al* [42] propose an approach that maps overlay peers

into individual points in Euclidean space and approximate the distances in IP infrastructures using Euclidean distances. Other work proposes to incrementally improve peer latencies by keeping a list of shortcuts in the routing table. The shortcuts generally point to nodes with smaller latencies. These goals are achieved either through interest-based locality [43], or through random sampling techniques [44]. All of the above work only consider network connectivities and may require extra services, such as node landmarks.

Several efforts have shown [45], [24, 46, 47] that to meet the applications' end-to-end QoS requirements, we need knowledge of real-time task information including the task's deadline, resource requirements and execution times. Most of them have shown that least laxity scheduling is an effective algorithm for distributed scheduling in soft real-time distributed systems.

6 Conclusions

In this paper we have proposed two self-organization algorithms that improve task execution times and system scalability in P2P systems. When a peer is discovered to frequently provide good execution times, the peer-based algorithm attempts to connect directly to that peer. If an underutilized peer discovers slow or overutilized processors, it attempts to move closer to those peers to improve the task execution times and balance the load across multiple processors. The experimental results show that our self-organization algorithms can effectively reduce the task end-to-end execution times, improve task miss ratio, and are able to dynamically adapt to changes in resource availability or peer connections and disconnections.

Acknowledgements. This work is supported by NSF Award 0330481.

References

1. Napster, "Napster home page," http://www.napster.com.
2. Gnutella, "Gnutella home page," http://www.gnutella.com.
3. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage," in *Proceedings of ASPLOS*, Cambridge, MA, 2000.
4. A. Rowstron and P. Druschel, "Storage Management and Caching in PAST, a Large-scale Persistent Peer-To-Peer Storage Utility," in *Proceedings of the 18th SOSP*, Toronto, Canada, 2001.
5. I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proceedings of ACM SIGCOMM Conference*, San Diego, CA, August 2001.
6. L. Xiao, X. Zhang, and Z. Xu, "On reliable and scalable peer-to-peer web document sharing," in *Proceedings of the International Parallel and Distributed Computing Symposium*, Fort Lauderdale, Florida, April 2002.

7. SETI Project Home Page, "SETI@home," http://sethiathome.ssl.berkeley.edu.
8. Entropia, "Entropia home page," http://www.entropia.com.
9. S. Jain, R. Mahajan, D. Wetherall, and G. Borriello, "Scalable self-organizing overlays," in *Technical report UW-CSE 02-02-02, University of Washington*, 2002.
10. Y-H Chu, S. G. Rao, and H. Zhang, "A case for end system multicast," in *ACM SIGMETRICS'00*, Santa Clara, CA, 2000.
11. D. Zeinalipour-Yazti, V. Kalogeraki, and D. Gunopulos, "Exploiting locality for scalable information retrieval in peer-to-peer systems," in *Information Systems Journal*, 2004.
12. C. Gill, J. P. Loyall, R. E. Schantz, M. Atighetchi, J. M. Gossett, D. Corman, and D. C. Schmidt, "Integrated Adaptive QoS Management in Middleware: A Case Study," in *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2004.
13. Object Management Group, "The Common Object Request Broker: Architecture and Specification," Edition 2.4, formal/00-10-01, October 2000.
14. D. Box, *Essential COM*, Addison-Wesley, January 1998.
15. A. Wollrath, R. Riggs, and J. Waldo, "A distributed object model for the Java system," *Computing Systems*, vol. 9, no. 4, pp. 265–290, Fall 1996.
16. SOAP, "Soap home page," http://www.soap.org.
17. V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser, "Dynamic scheduling for soft real-time distributed object systems," in *Proceedings of the IEEE Third International Symposium on Object-Oriented Real-Time Distributed Computing*, Newport, CA, March 2000, pp. 114–121.
18. H.-H. Chu and K. Nahrstedt, "A soft real-time scheduling server in unix operating system," 1995, pp. 381–406, Auerbach Publications.
19. Object Management Group, "Real-time CORBA," Edition 1.0, formal/00-10-60, May 1998.
20. Object Management Group, "Dynamic Scheduling," Revised Submission, orbos/00-08-12, August 2000.
21. Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny, "Condor – a distributed job scheduler," in *Beowulf Cluster Computing with Linux*, Thomas Sterling, Ed. MIT Press, October 2001.
22. Vana Kalogeraki, *Resource Management for Real-Time Fault-Tolerant Distributed Systems*, Ph.D. thesis, University of California, Santa Barbara, Dec. 2000.
23. V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser, "Dynamic scheduling of distributed method invocations," in *Proceedings of the 21st IEEE Real-Time Systems Symposium*, Orlando, Florida, November 2000, pp. 57–66.
24. M. L. Dertouzos and A. K.-L. Mok, "Multiprocessor on-line scheduling of hard real-time tasks," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497–1506, December 1989.
25. The FFMPEG Homepage, "http://ffmpeg.sourceforge.net/," .
26. A. Mohan and V. Kalogeraki, "Speculative Routing and Update Propagation: A Kundali Centric Approach," in *International Conference on Communications*, Anchorage, Alaska, May 2003.
27. K. Aberer, M. Punceva, M. Hauswirth, and R. Schmidt, "Improving Data Access in P2P Systems," *IEEE Internet Computing*, vol. 6, no. 1, pp. 58–67, January/February 2002.
28. V. Kalogeraki, A. Delis, and D. Gunopulos, "Peer-to-Peer Architectures for Scalable, Efficient and Reliable Media Services," in *Proceedings of the International Parallel and Distributed Computing Symposium*, Nice, France, April 2003.

29. S. Waterhouse, D.M. Doolin, G. Kan, and Y. Faybishenko, "Distributed Search in P2P Networks," *IEEE Internet Computing*, vol. 6, no. 1, pp. 68–72, January/February 2002.
30. R. Lienhart, M. Holliman, Y-K. Chen, I. Kozintsev, and M. Yeung, "Improving Media Services on P2P Networks," *IEEE Internet Computing*, vol. 6, no. 1, pp. 73–77, January/February 2002.
31. V. Kalogeraki and F. Chen, "Managing distributed objects in peer-to-peer networks," *IEEE Network, special issue on Middleware Technologies for future Communication Netowkrs*, vol. 18, no. 1, pp. 22–29, January 2004.
32. Morpheus, "Morpheus home page," http://www.musiccity.com.
33. Freenet, "Freenet home page," http://freenet.sourceforge.com.
34. Kazaa, "Kazaa home page," http://www.kazaa.com.
35. Limewire, "Limewire home page," http://www.limewire.com.
36. S. Ratnasamy, P. Francis, M. Handley, and R. Karp, "A Scalable Content-Addressable Network," in *Proceedings of the SIGCOMM'01*, San Diego, CA, August 2001.
37. Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam, "Taming Aggressive Replication in the Pangaea Wide-Area File System ," in *Proceedings of OSDI 2002*, Boston, CA, 2002.
38. A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," in *Proceedings of OSDI 2002*, Boston, CA, 2002.
39. A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A Read/Write Peer-to-Peer File System," in *Proceedings of OSDI 2002*, Boston, CA, 2002.
40. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Topologically-aware overlay construction and server selection," in *Proceedings of IEEE INFOCOM Conference*, June 2002.
41. Z. Xu, M. Mahalingam, and M. Karlsson, "Turning heterogeneity into an advantage in overlay routing," in *Proceedings of IEEE INFOCOM Conference*, April 2003.
42. T.S. Eugene and H. Zhang, "Predicting Internet Network Distance with Coordinates-based Approaches," in *Proceedings of IEEE INFOCOM Conference*, 2002.
43. K. Sripanidkulchai, B. Maggs, and H. Zhang, "Efficient Content Location using Interest-based Locality in Peer-to-Peer Systems," in *Proceedings of IEEE INFOCOM Conference*, April 2003.
44. H. Zhang, A. Goel, and R. Govindan, "Incrementally improving lookup latency in distributed hash table systems," in *Proceedings of ACM SIGMETRICS Conference*, 2003.
45. G. Manimaran and C. R. R. Murthy, "An efficient dynamic scheduling algorithm for multiprocessor real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 3, pp. 312–319, March 1998.
46. F. Sandrini, F. D. Giandomenico, A. Bondavalli, and E. Nett, "Scheduling solutions for supporting dependable real-time applications," in *Proceedings of the IEEE Third International Symposium on Object-Oriented Real-Time Distributed Computing*, 2000.
47. J. Hildebrandt, F. Golatowski, and D. Timmermann, "Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems," in *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99.*