# Performance Information Sharing Infrastructure

CHARLES REISS

Abstract: This thesis presents a design for distributed monitoring system designed to enable monitoring-informed optimizations in distributed applications. Microbenchmarks and an evaluation in a scientific-computing scenario are presented. The monitoring system is intended to assist when application requirements cannot be easily expressed in a form suitable for existing autonomic computing approaches. The design embeds awareness of the application's topology into the monitoring system so queries can reference a node's place in the application without embedding extra assumptions about the overall layout of the application. Through integration with dynamic code generation, users may make potentially application-specific metadata available and use such data within dynamically deployed filters and transformation functions. Evaluations demonstrate that this approach can provide timely and useful information with low overhead.

## 1. INTRODUCTION

Middlewares find feedback from live measurements useful in tweaking resource allocations. Previous work has tuned operator placement, service and path choice based on user utility functions and real-time performance measurements. Similar metrics should inform lower-level decisions. While many prior systems support the aggregation of performance information for monitoring, error identification, and load balancing, such systems are not easy for distributed applications to use when their performance heuristics do not fit into traditional autonomic-computing models.

This thesis describes a monitoring system designed to support in-application tuning. Through integration with out middleware, applications can easily retrieve and act upon measurements based on the path of their data. Users can exploit these measurements in not only existing application code, but also through dynamically deployed filters and transformations functions placed anywhere along the data flow.

The design we present can implement policy requirements that prior systems handle poorly. We evaluated one such scenario: scientists running a large simulation often wish to collect a large amount of data for later analysis while examining some diagnostic output to verify the correctness of their simulation. Because early detection of errors saves expensive CPU time, these users want as high quality, timely

diagnostic data as possible without sacrificing bandwidth required to send the large datasets they wish to analyze. Ideally, users would reduce their data flow based on available capacity. However, because extensive buffering and intermediate transfer nodes hides the latency of transfer, the information required to adjust dataflow is not easily made available near the source. Our system will allow filters reducing that data flow to be written with little knowledge of the downstream dataflows.

## 2. BACKGROUND AND LITERATURE REVIEW

### 2.1 Network Monitoring and Querying

Many researchers have focused on monitoring large networks to make performance predictions and report on overall network health. When aimed at Internet-scale networks, these networks both passively measure ordinary traffic from a sample of nodes [Padmanabhan et al. 2005; Zhang et al. 2004; Madhyastha et al. 2006]. By recording the timing of data flowing past these nodes can yield latency and bandwidth information as well as notification of extraordinary conditions like routing path changes. To gather more complete path information and coverage, these monitoring systems choose a representative sample of hosts to probe explicitly.

On more centrally controlled networks, monitoring and aggregation systems tend to be built more directly around supporting distributed applications. SDIMS [Yalagandula and Dahlin 2004] and Astrolabe [Van Renesse et al. 2003], for example, are intended to support application queries. These systems essentially act as distributed databases which gather the results of potentially continuous queries over a distributed application. These systems can be thought of as a high-level middleware that will, in a robust manner, support application coordination tasks, such as load balancing, replica discovery, and overall application monitoring. Some similar, lighter-weight systems that target administrator queries like Ganglia [Massie et al. 2004] have similar designs though their aggregation tasks are simpler.

### 2.2 Distributed Databases

A great deal of research has created scalable and potentially widely distributed databases, which can operate in the absence of centralization. One prominent approach to distributed databases has been distributed hash tables (DHTs). Because of its scalability and robustness, DHTs are the basis of the distributed monitoring system SDIMS [Yalagandula and Dahlin 2004]. Distributed hash tables arrange

nodes in a ring based on the randomly assigned addresses [Stoica et al. 2003]. Given a key, DHTs can route messages to the node closest to without exceeding that key efficiently through its overlay network. Given this, storage and aggregation applications are easily supported in a scalable manner. With careful replication and route management, distributed hash tables can provide reasonable performance such applications even over widely distributed networks with a great deal of churn (that is, nodes entering and leaving the network) [Rhea et al. 2004]. Distributed hash tables tend to assume total connectivity among nodes and do not take advantage of locality, as is desired in typical aggregation situations. Fortunately, research has modified DHTs to deal with incomplete connectivity [Freedman et al. 2005] and to create sub-rings based on network locality [Freedman and Mazires 2003].

### 2.3 High-Performance Data Movement

Data extraction in high-performance systems is a challenging task due to the amount of data produced. A large simulation will soon produce petabytes a week, easily saturating available bandwidth to move the data from the simulation site to longer-term storage for analysis. Using multiple simultaneous data streams and intermediate storage repositories, prior systems efficiently transfer this data across wide-area links [Bhat et al. 2004; Allcock et al. 2002].

Along with their data extraction tasks, many simulation users wish to analyze and visualize their data as it is being produced. Such visualization tasks can efficiently be performed by rendering subsets of the data in pieces near the simulation source and compositing these images closer to the viewer [Ma and Camp 2000; Yu et al. 2004].

To allow scientists to easily specify their data analysis and transfer tasks, researchers have deployed workflow systems like Kepler and Pegasus [Ludscher et al. 2005; Deelman et al. 2004]. Workflow systems are essentially visual programming environments which allow their users to connect a sequence of independent tasks together and then assign those tasks to machines. After the user has programmed their scenario, the workflow systems then actually schedules the tasks.

Typically workflow systems only coordinate actors that operate at file granularity. Similar, more fine-grained support for combining components does, however, exist in middlewares that support stream-like abstractions [Kumar et al. 2006]. Such middlewares enable generic, realtime manipulation of data in transit. For example, such support can be used to implement generic operations such as

compression that adjusts to optimize the space/CPU tradeoff [Wiseman et al. 2004]. When the middleware has type knowledge to treat the stream as a series of typed atoms, it may support more complex, data-aware manipulations [Eisenhauer et al. 2001]: if, for example, such middlewares allow data filtering to be performed independent of an existing application to reduce the transferred data volume [Lofstead and Schwan 2005]. With runtime code generation, datatype-dependent filtering can be deployed without changing or restarting the augmented application.

## 3. DESIGN

### 3.1 Middleware Integration

Many applications are easily expressed in terms of their network data flow. Our group's middleware, like many middlewares of this sort, represents applications as data-carrying connections between calculations. We built our prototype on top of the EVPath middleware, which is a derivative of ECho [Eisenhauer et al. 2001]. In our middleware's terminology, the application is divided into stepping 'stones' at which user-registered actions manipulate the data and possibly pass it onwards to other stones. At each stone runtime-generated and possible runtime-deployed code may make transformation and routing decisions. The measurement system presented in this thesis integrates with our middleware, so users can more naturally publish and receive information within their applications.

By integrating with this middleware, the system preserves the flexibility of runtime operator placement. Calls to our measurement infrastructure through runtime-generated code are implicitly annotated with topology information. Since the topology information is not derived from the code itself, it may be freely redeployed around the network and still benefit from nearby measurements. Even if users never move or duplicate operations, users should be be able to avoid coupling their overlay network topology to lower-level optimization code.

### 3.2 The Distributed Database

Published measurements are made available through a distributed database. Each datum published by the application is stored with the stone with which it is associated and its expiration time. The actual data published is marshaled using our middleware and thus may be of any type the middleware can transfer. Currently, the database is an instance of the Chord distributed hash table [Stoica et al. 2003], but this

choice should be inconsequential. Each machine in our application's overlay network runs a DHT node, which is used to both publish and query the database. Future work could explore using a distributed database that is aware of the application's topology when it places data.

Caching and asynchronous database queries assure that overhead remains low. Our system caches database entries and full query results based on the stored expiration time. Since measurements provided should primarily serve as performance hints, stalling the application while waiting for data would be counterproductive. So, rather than blocking when a sufficiently current measurement is not available, our library returns expired cached data or no data and retrieves the current entry asynchronously.

### 3.3 Publishing Measurements

Applications can synthesize their measurements by providing a callback that can generate it on demand or by explicitly pushing values out from processing functions called by our middleware. Measurements explicitly pushed out from filters and transformation functions automatically capture the node with which they are associated. These passive measurements will encompass information that is already present within the application but not in a memory space where it would be useful. Actively published measurements encompass the other case, where the measurement must be synthesized. To publish measures actively, the user supplies a small piece of code which generates the measurement's value. This code is run for each node or machine (depending on the type of measurement) at a frequency specified by the user.

As the number of available measurements increases, publishing this many measurements may produce substantial overhead. For measurements generated by registered callbacks, using these callbacks can be deferred until the datum is needed. Instead of generating such callbacks automatically, an entry with longer lifetime can be published in the distributed database subscribing to requests for interest in that datum. When a node starts querying the datum, it would notify the publishing node through the distributed database, triggering uses of that callback.

For measurements published in data filters that use our C-like mini-language, we could optimistically assume that the application will not query a passive measurement. Thus, we could initially omit publishing measurements when compiling filters, making note of what would keys have been published. When, as with callback-registered measurements, production of the measurement is triggered, the compiled code for filter in question can be invalidated. When the filter is recompiled, the middleware

no longer elides the measurement generating code. Although this thesis does not include tests of situations where publishing-eliding is helpful, these techniques should ensure overhead is proportional to how much data the application uses.

### 3.4 Using Topological Awareness

Measurements from upstream and downstream in the logical flow of data are among the most useful in distributed applications. Such measurements are those that a filter or transformation can most easily affect. Even without knowing the purposes of the ultimate sources and sinks for their data, filters and transformations can use generic performance measurements, such as of CPU usage and buffer space usage, since they result from or will affect their dataflow. Such relationships are probably more useful when the application developer can take the time to use more application-specific data. Thus, we allow users to query performance measurements solely based on their relative location within the dataflow graph: the user can request data from nodes or machines within a certain number of steps downstream or upstream.

To support such queries, we instrument the middleware to export the middleware's overlay topology into the distributed database. Each stone's potential paths to other stones, remote or local, is recorded in the distributed database. Any machine on the network walk the path of data through the network by querying the database recursively. Since we anticipate that the application topology is relatively stable, topology information can be cached aggressively, hiding the overhead of these recursive queries in typical cases. By acting on the level of the middleware's abstraction and not physical machines or protocol-level paths, the queried information separates data paths that cross the same machine. The information may be unreliable during reconfiguration and fail to discount rarely used links, but it should be reliable often enough for most practical uses.

### 3.5 Summarizing Query Results

In large systems, a query could easily require information from dozens of nodes that must be reduced into a coherent query answer. Clearly there is a need for different types of summaries in different cases, even when querying the same data: for example, to determine how timely data delivery is, a user would want to know the total used buffer space downstream but to determine whether nodes are overloaded, a user would query the maximum free buffer space of any one node. Flexibility is provided by allowing users to specify a function that will combine a partial answer with additional data, which form part of the query. This

functionality can reuse our existing dynamic code generation rather than requiring a separate query language, and it should still allow query computation to be distributed in the future.

## 4. EVALUATION

### 4.1 Microbenchmarks

For our framework to be useful, publishing and querying must have little overhead. To ensure wide deployment, the overhead when no information is queried must be also be minimal. Since the cost of maintaining the distributed database should be very small in absolute terms and compared to the cost of normal traffic and processing, most of the cost of using and producing measurements should result from communication latency. Thus, almost all of the overhead of querying and publishing should be hidden from the application by asynchronous access at the expense of lower information quality. But since our system caches database entries, long cache expiration times should usually be the largest contributor to stale data. Users should be able to increase information freshness by decreasing expiration times, at the expense of increased overhead.

To measure the overhead of our infrastructure, we ran a simple two-machine experiment. Two machines, connected by a high-speed link, sent, as fast as possible, a stream of 64-kilobyte datums. Each datum was annotated with a sequence number that in non-control runs was published through our infrastructure at the source. The sink read these sequence numbers and compared them with the datum at the time it was ready for processing. Several versions of this experiment were run with varying expiration times (and thus publishing frequencies) for the sequence numbers. Additionally a control run was used in which no data was published or queried.

The difference in transmission rates between control and non-control experiments measured overhead in terms of available bandwidth. This overhead was negligible (less than 0.1%), with nearly all of a hundred megabit Ethernet link in use. This suggests that overhead in terms of bandwidth is very small. A similar experiment on Infiniband suggested that our system was similarly efficient in terms of CPU overhead. Without and without our system active we maintained transmission rates of around 1890 megabits per second (after protocol overheads). Any drop in transmission rates when our system was running was less than 10 megabits, less than what the experimental setup could reliably measure. Thus, we can infer that effective CPU overhead was well less than 1%. Our experiments, were, however run on a

dual-core which may have hidden some of the CPU overhead by making effective use of the second core.

From these experiments, we also inferred the freshness of metadata. Using the rate of transmission information and the difference between sequence numbers read through our infrastructure, the latency of published data was computed. Latency caused by link speed and protocol overhead was also measured, but was insignificant for this purpose (less than eight milliseconds). As expected, the overall latency was primarily determined by the periodicity and expiration time of the published data: the measured latencies were approximately one second larger than half the expiration time. The standard deviation of the latency was approximately 5 seconds for small expiration times and approximately equal to the latency for larger expiration times.

### 4.2 Wide-area Evaluation

Wide-area systems need to adapt to changes in network performance. Since available bandwidth is the scarce resource in wide-area data transmission, backlogs manifest themselves in, among other problems, excess buffer usage along the data path. Retrieving information about application buffers is easy, and over the long term, this information should reflect whether the current data transmission rate is feasible. We used this information to balance the transmission of two data streams, each with different performance demands.

Our test included retrieving two different forms of output from the Gyrotordial Collider Code, a large plasma physics simulation: from a 32-node simulation run, we extracted the magnetic potential grid with sufficient metadata to visualize it. A filter was deployed at the site at which data left the simulator which reduced the data volume by taking a sample of the grid points based on a parameter read through our system. This number was synthesized by querying the size of buffers (in terms of numbers of atoms) downstream of the filter. When this size was increasing significantly, the reduction factor was increased, when it was decreasing significantly or had stayed at 0 over several samples, the reduction factor was decreased.

The simulation was configured to output the magnetic potential grid every time step. The time between timesteps varied between 20 and 30 seconds, and approximately eight megabytes of data was produced in each timestep. After the data was collected at the filtering site it was sent through a high-latency TCP stream simulated using NS-2 which had an effective bandwidth of 200 kilobits per

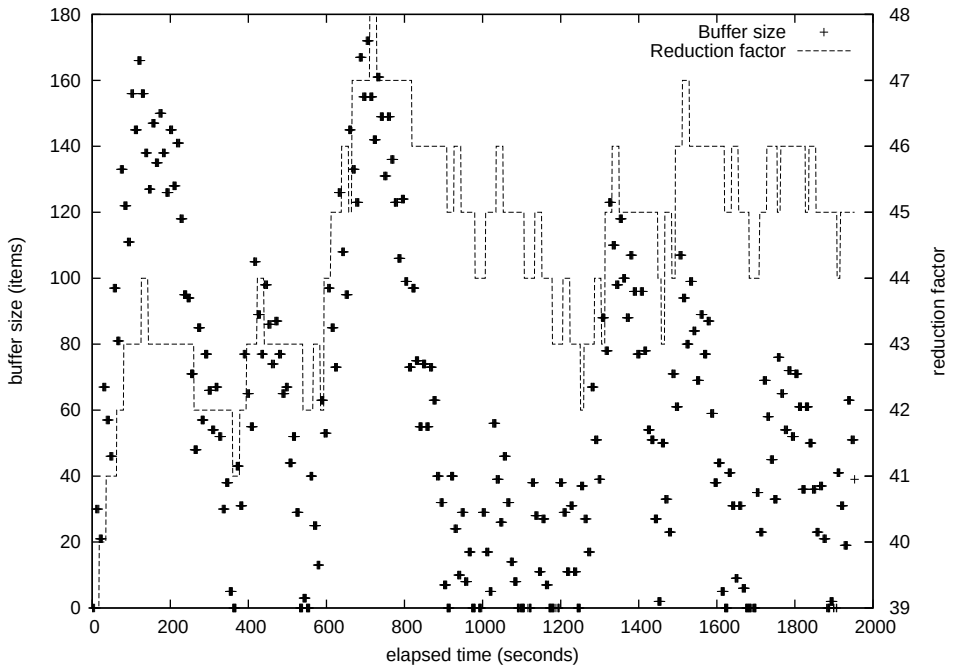


Fig. 1. Observed buffer sizes and corresponding reduction rates chosen over a single GTC run

second. In spite of the substantial variation in data transfer rates and the high latency, using this very simple feedback loop we successfully maintained good utilization of the link and relatively low effective data latency.

## 5. CONCLUSION

This thesis presents a design for a viable system for sharing heuristic hints within distributed applications. Microbenchmarks and the scientific visualization inspired evaluation demonstrate that it is a viable and low-overhead approach. By creating a distributed monitoring system that is aware of the application's topology, we show that this information can be shared without need for ad-hoc arrangements that would otherwise be required to arrange for its transfer. Simultaneously, it avoids the central coordination that is often present in traditional autonomic systems. Similarly, by not requiring application developers to reduce their complex requirements to global utility functions and their monitoring-dependent to simple operations like operator placement, this design can handle complex or unusual user requirements.

## REFERENCES

Allcock, B., Bester, J., Bresnahan, J., Chervenak, A. L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., and Tuecke, S. 2002. Data management and transfer in high-performance computational grid environments.

*Parallel Computing 28,* 5, 749–771.

Bhat, V., Klasky, S., Atchley, S., Beck, M., McCune, D., and Parashar, M. 2004. High performance threaded data streaming for large scale simulations. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*. 243–250.

Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M. H., Vahi, K., and Livny, M. 2004. Pegasus: Mapping scientific workflows onto the grid. In *Across Grids Conference*. Nicoasia, Cyprus.

Eisenhauer, G., Bustamante, F. E., and Schwan, K. 2001. A middleware toolkit for client-initiated service specialization. *ACM SIGOPS Operating Systems Review 35,* 2, 7–20.

Freedman, M. J., Lakshminarayanan, K., Rhea, S., and Stoica, I. 2005. Non-transitive connectivity and DHTs. In *Proceedings of the Second Annual Workshop on Real, Large, Distributed System (WORLDS '05)*. San Fransciso, California.

Freedman, M. J. and Mazires, D. 2003. Sloppy hashing and self-organizing clusters. In *Proceedings of the Second Intl. Workshop on Peer-to-Peer Systems (IPTPS '03)*. Berkeley, CA.

Kumar, V., Zhongtang, C., Cooper, B. F., Eisenhauer, G., Schwan, K., Mansour, M., Seshasayee, B., and Widener, P. 2006. Implementing diverse messaging models with self-managing properties using iflow. In *3rd IEEE International Conference on Autonomic Computing*. Dublin, Ireland, 243–252.

Lofstead, J. and Schwan, K. 2005. Xchange: High performance data morphing in distributed applications. Tech. rep., Georgia Institute of Technology.

Ludscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao, J., and Zhao, Y. 2005. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice & Experience*.

Ma, K. L. and Camp, D. M. 2000. High performance visualization of time-varying volume data over a wide-area network. In *ACM/IEEE 2000 Conference on Supercomputing*. 29–29.

Madhyastha, H. V., Isdal, T., Piatek, M., Dixon, C., Anderson, T., Krishnamurthy, A., and Venkataramani, A. 2006. iPlane: an information plane for distributed services. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Seattle, Washington, 367–380.

Massie, M. L., Chun, B. N., and Culler, D. E. 2004. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing 30,* 7, 817–840.

Padmanabhan, V., Ramabhadran, S., and Padhye, J. 2005. Netprofiler: Profiling wide-area networks using peer cooperation. In *Proceedings of the Fourth International Workshop on Peer to Peer Systems (IPTPS)*, M. Castro and R. v. Renesse, Eds. Ithica, New York, 80–92.

Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. 2004. Handling churn in a dht. In *Proceedings of the 2004 USENIX Technical Conference*.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. 2003.

Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON) 11,* 1, 17–32.

Van Renesse, R., Birman, K. P., and Vogels, W. 2003. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems 21,* 2, 164–206.

Wiseman, Y., Schwan, K., and Widener, P. 2004. Efficient end to end data exchange using configurable compression. In *Proceedings of the 24th International Conference on Distributed Computing Systems*. 228–235.

Yalagandula, P. and Dahlin, M. 2004. A scalable distributed information management system. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Portland, Oregon.

Yu, H., Ma, K.-L., and Welling, J. 2004. A parallel visualization pipeline for terascale earthquake simulations. In *Proceedings of the ACM/IEEE Supercomputing Conference*, J. Welling, Ed. Pittsberg, Pennsylvania, 49–49.

Zhang, M., Zhang, C., Pai, V., Peterson, L., and Wang, R. 2004. PlanetSeer: internet path failure monitoring and characterization in wide-area services. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*. Vol. 6.