AggMon: Scalable Hierarchical Cluster Monitoring

Erich Focht and Andreas Jeutter

Abstract Monitoring and supervising a huge number of compute nodes within a typical HPC cluster is an expensive task. Expensive in the sense of occupying bandwidth, and CPU power that would be better spend for application needs. In this paper, we describe a monitoring framework that is used to supervise thousands of compute nodes in a HPC cluster computer in an efficient way. Within this framework the compute nodes are organized in groups. Groups contain other groups and form a tree-like hierarchical graph. Communication paths are strictly along the edges of the graph. To decouple the components in the network a publish/subscribe messaging system based on AMQP has been chosen. Monitoring data is stored within a distributed time-series database that is located on dedicated nodes in the tree. For database queries and other administrative tasks a synchronous RPC channel, that is completely independent of the hierarchy has been implemented. A browser-based front-end to present the data to the user is currently in development.

1 Introduction

One of the often overlooked challenges in modern super-computing is the task to track system state, supervise compute node status and monitor job execution. As node counts increase, monitoring becomes a task that consumes a significant amount of network bandwidth and CPU power.

The development and investigation described in this paper were done as part of the TIMACS project and were sponsored by NEC and the German Bundesministerium für Bildung und Forschung.

E. Focht (✉) · A. Jeutter
NEC HPC Europe, Hessbrühlstr. 21b, 70565 Stuttgart, Germany
e-mail: efocht@hpce.nec.com; ajeutter@hpce.nec.com

Thus challenges for a HPC cluster monitoring system are:

- Minimise communication demands: bandwidth should be preserved for the application jobs.
- Scalability: keep growth-rate of infrastructure demands of the monitoring system well below the growth-rate of total compute nodes in the system.
- Minimise CPU usage: run as a subordinate task on the compute nodes but propagate critical system states as fast as possible.
- Fault tolerant and self recovery: a single failure of a compute node should not cause the monitoring system to collapse.

2 Previous Work

One of the first common and widely used tools to monitor large scale cluster hardware was the Berkeley University developed Ganglia [2]. Ganglia uses a distributed architecture approach and utilize unicast or multicast communication to send monitoring data to a master node. A configurable front end application displays the data in various ways and provides an overview of the whole system.

Van Renesse et al. developed and described in [10] an information management system that collects monitoring data and tracks system state on large computing sites. This system uses an hierarchical approach where compute nodes are put in zones. Zones are organized in a hierarchical fashion where each zone aggregates its data in relatively small portions to leverage bandwidth.

Marsh et al. investigated in [9] into scalability, reliability and fault tolerance of AMQP messaging systems. They proposed a federation hierarchy of nodes in conjunction with a dedicated configuration that is based on experimental data to gain maximum scalability.

Wang et al. described in [11] a messaging system using a publish/subscribe mechanism to send information over a distributed system. They added features to priorize topics and thus gained real-time performance for critical messages.

3 Architecture and Design

3.1 Core Design Decisions

The core design decisions for AggMon were driven by the target of reaching high scalability of the monitoring infrastructure while keeping the network as lightly loaded with monitoring data, as possible. Aiming at specialized HPC machines with huge numbers of compute nodes we consider a fixed or at least very slowly changing monitoring hierarchy to be a very realistic approach. O(1000–10000) specialized compute nodes deserve a hierarchy of dedicated administration nodes that take over the load of monitoring and keep as much as possible of it away from the compute nodes and compute network.

Scalability of the communication infrastructure for monitoring data is rarely addressed, but its choice can influence the way how to deal with increasing numbers of data reporters and temporary outages of the network or of administration servers. We decided for topic based publish/subscribe (eg. [8]) semantics. They allow for a nice asynchronous design of communicating components, the data producers can send out their data and forget about them, while data consumers can register handlers for the particular data topics they will be processing.

For collecting monitoring data we don't want to re-invent the wheel. Instead we want to be flexible and use data collected by already existing monitoring components like ganglia, nagios, collectd, collectl. In order to keep the traffic of monitoring data limited we use data aggregation heavily and only push aggregated data representing meaningful information about a group's state upwards the hierarchy tree.

The current value and time history of metric data is stored in distributed manner spread across the administration nodes, keeping the compute nodes free of the burden of disk I/O for monitoring data.

3.2 Hierarchy

In order to improve the scalability and manageability of huge computer systems a distribution of data and load is needed. Two approaches seem natural: the use of peer-to-peer and overlay networks or the use of a hierarchy or even combine both ideas like in [10].

We decided for a rather static hierarchy and against the use of overlay networks because for HPC systems the compute nodes should be kept free of any additional load which could spoil the scalability of the user applications. Therefore compute nodes should at most take care of generating their own monitoring data and sending it upstream the hierarchy path, but not need to "know" about monitoring data of other compute nodes or even try to aggregate data in any way. The hierarchy consists of groups and client nodes. Groups can contain groups or clients and must have at least one master. Group masters must not necessarily be located inside the group they are responsible for. The top level group is called "universe" and contains all groups and client nodes of the system. This hierarchy that can be represented as a direct acyclic graph can be compared to a UNIX filesystem hierarchy where groups correspond to directories and client nodes and masters correspond to files. The root path "/" corresponds to the "universe" group and a list of full file paths would describe all nodes and groups in the hierarchy.

Figure 1 shows a simple hierarchy consisting of eighteen compute nodes plus six master nodes spread over five groups. Each node in the system can be described by a unique path, for example:

$$/b/x/n07$$

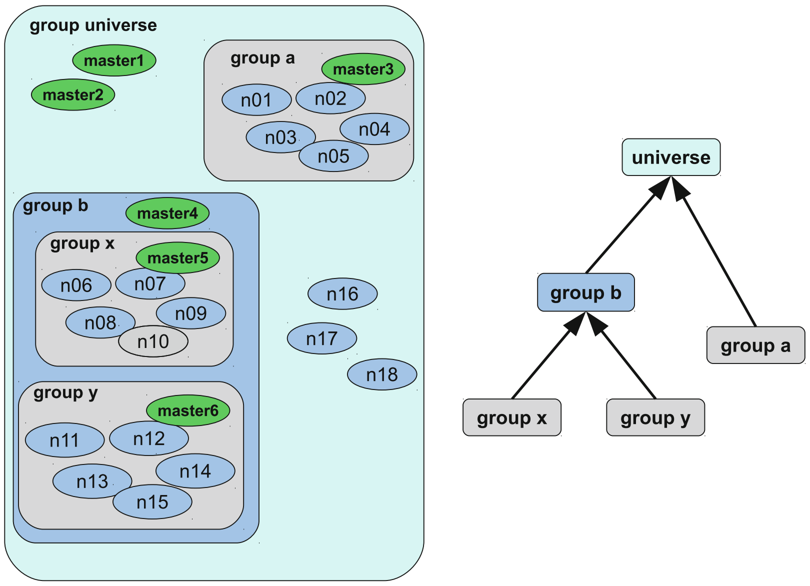

Fig. 1 Group hierarchy consisting of five groups where the groups *x*, *y* are contained in group *b*, and the groups *a* and *b* are contained in the group *universe*. The group dependency *graph* on the *right side* of the figure corresponds to the flow of group-aggregated monitoring data

is the full group path of node *n07*. It is a direct member of group *x*, which itself is member of group *b*. Each group's master node collects the monitoring information of its own nodes and of its subgroups. Time series information is kept on the group master nodes, aggregated monitoring data is pushed up the hierarchy tree to the higher level group's master nodes.

3.3 Components

The components built into the AggMon daemon are depicted in Fig. 2.

The core that links all components is the *channel*. It provides a topic based publish/subscribe abstraction. Channels can be opened, subscribed and published to. They are addressed with a URI.

The primary source of monitoring data are *importers*. They can run on compute nodes and collect local monitoring data or they can run on the group master nodes and collect metrics from the compute nodes. *Importers* publish the measured metrics to their group's master node with a topic that contains information on the data source, the data type and the originating host.

Each node that acts as a group master runs the *database* component. It subscribes to the group's channel and "sees" each metric published within the own group.

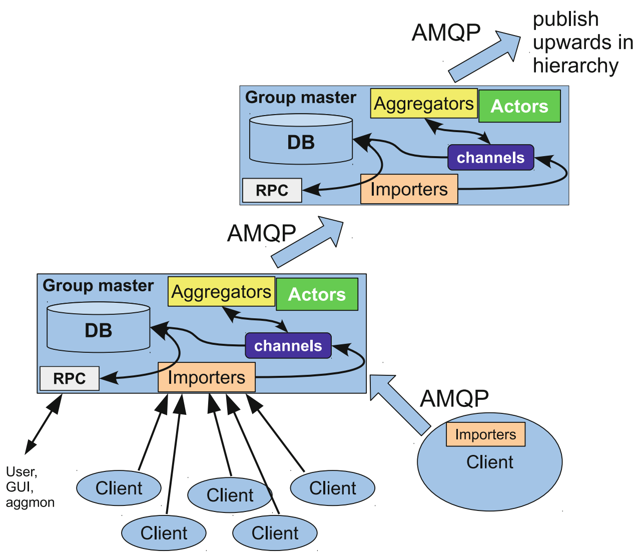

Fig. 2 Architecture or AggMon: Client nodes metrics are either published by importers running on clients or gathered by importers running on group master nodes. Group master nodes run various services like Database, Channels, Aggregators, and publish aggregated group data upwards in the monitoring hierarchy

Metrics are serialized and stored to disk. The time history of the monitoring data is also stored within the database. Each database instance keeps only information of its own group's nodes and subgroups. All group master nodes together form a specialized distributed monitoring database.

An *RPC* server component serves as user interface to the database component, allows querying the stored metrics and time-series information in synchronous way. Furthermore the RPC component is used to query and control the status of the AggMon daemon.

Aggregators are subscribers to the own group's channel and generate new metrics by using the information they see published by the group's members (nodes and subgroups). They can generate new host metrics (for example, compute number of fans running in a node out of the separately measured fan speeds) which are published on the group's channel as if they were measured by a normal importer. Or they can generate group metrics using any metrics from a group, on behalf of the group (for example, the highest node temperature in a group) and publish it upstream to the higher group's master.

Finally, *actors* are configured to execute commands initiated and triggered by aggregators or by the RPC component. This way an email could be sent out or an emergency script could be run if an aggregator considers that it discovered a problem that it should react to. Actors are planned but were not yet implemented at the time when this paper was written.

4 Implementation

AggMon is entirely written in the Python programming language [5] which allows fast prototyping. The language features a huge standard library with modules that can be loaded at runtime. The language definition is publicly available and currently several runtime environments are available. Python is very popular and supported by a large community of developers.

4.1 Publish/Subscribe

Message Brokers and Channels

All monitoring components use a topic-based publish/subscribe infrastructure to exchange the measured monitoring metrics. The components of AggMon act either as publishers, as subscribers or in some cases as both. Importers, the measurement components, collect data and publish it under particular topics on the network. Databases and aggregators act as subscribers to particular topics, aggregators publish the derived metric values.

The *channels* abstraction in AggMon is providing publish/subscribe semantics to the components. A channel is being addressed with a URL, its most generic form being:

```
<protocol>://[username:password@]hostname/channelname
```

Protocol is the used underlying implementation and can be one of:

- **local:** an optimization allowing threads local to a daemon process to exchange messages with topic based publish/subscribe.
- **amqp:** uses an external AMQP broker and is implemented with the py-amqplib library [4].
- **pika:** also uses an external AMQP broker, is implemented on top of the pika python library [3].

Username and password are used for authenticating with the message transport layer and are needed only when using one of the AMQP protocols.

The underlaying Advanced Messaging Queueing Protocol (AMQP, [1]) network relays the messages from the publisher to the subscriber and works as a message broker that can buffer data and decouples data generation and consumption. Its asynchronicity is very beneficial for scalability. AMQP has its origins in financial applications which deal with high numbers of transactions and strict requirements on availability and fault tolerance.

The AMQP broker is a component that should be considered a part of the underlaying network layer and is not subject to modification. The broker accepts messages that it receives from publishers. For each topic the broker maintains an internal queue where it appends the newly arrived message. Then the broker calls all subscribers for that particular topic and delivers the queued messages.

Usual brokers can run on single nodes but can also spread over networks and run on many nodes and build a big virtual clustered broker. Messages are automatically routed between broker instances to the ends where they need to be delivered to subscribers.

For our implementation we use the RabbitMQ messaging infrastructure [6], a widely used robust AMQP implementation written in the Erlang programming language.

Messages

The messages sent over the publish/subscribe infrastructure are JSON-serialized instances of the Metric class and consist of a set of key-value pairs. Some keys are mandatory:

- *name*: the metric name
- *source*: a hint on which component has created the metric
- *time*: the metric's measurement time
- *value*: the value of the measured entity.

Channel API

Applications can use the *channel* API to interface with the monitoring data network. The following code snippet sketches the usage of the three available API functions:

```
from aggmon.channel import *

# open a channel
channel = Channel.open( url, durable=False )

# subscribe to a topic, match group and metric name
topic = "group.*.*.metric_name"
channel.subscribe( topic, notify\_callback, raw... )

# publish a message on the channel
topic = "group.host.source.metric_name"
channel.publish( topic, message )
```

4.2 Importers as Metric Data Publishers

A publisher is a source of data it measures a physical value transforms and packs it into a Metric and publishes it under a certain topic in the network. The key point is that the publisher does not care and even does not know anything about the further treatment of the published data. One benefit of this scenario is that publishers can be separate short programs that are dedicated to a particular task. They even can sleep for longer periods and do not need to run permanently. One drawback of such asynchronicity is that common synchronous calls (RPCs, call and response pattern) are not possible. To overcome this the publisher can subscribe to a command topic and react on that. But most of the time this is not desirable. Another advantage of Public/Subscribe is that messages are guaranteed to be delivered. This is due to the possibility of the broker to store messages internally. Even if a subscriber is not running the messages are stored and delivered when the subscriber comes back online. Thus the grade of asynchronicity is only limited by the amount of data the broker can store. This provides fault tolerance and robustness.

4.3 Subscribers: The Metric Data Consumers

Subscribers are the contrary part of publishers. Subscribers connect to the broker and subscribe to a particular topic. The database and the aggregators are typical subscribers.

4.3.1 Database

The monitoring data is stored in a special purpose distributed database. Only group master nodes store data, they run an instance of the database for each group they represent.

The monitoring metrics are serialized to disk in a simple directory hierarchy. Each database instance for a particular hierarchy group path stores its data in an own directory. Each group member, host or subgroup, gets a subdirectory where each of its metrics is stored in a subdirectory of its own. Metrics attributes or metadata are stored in files named after each attribute. Time-series of metrics are abstracted into two classes: numerical records and log records and stored in separate files inside the metric subdirectory. Numerical records consist of the metric measurement time and value. Log records have string or unicode values and an additional optional "output" field. Currently each time-series file spans a certain time range, by default one day of data. In near future this will be extended by gradual thinning and averaging out of old data, in order to limit the amount of storage needed in a way similar to round-robin databases.

The database exposes several methods via RPC that can be called to retrieve data. All database instances are aware of the hierarchy and forward requests for data that is not available locally to remote database instances.

Database API

To query the database via RPC a connection must be opened to an arbitrary database instance. The query that would locally correspond to the call of a function

```
method(arg1, arg2, key3=arg3, ...)
```

is sent flattened as plain text in the form

```
method arg1 arg2 key3=arg3 ...
```

over the RPC channel. The database returns the results also as plain text that represents a valid Python object. This text can be evaluated via the *eval()* method to gain a Python object.

The concrete database API is still in development and being adapted and modified to the needs of programs that need to interact with the database, like a graphical user interface. Currently it consists of following functions:

- *dbSetOffline(group_path)*: Set current instance of database offline. In offline state the database doesn't commit received metrics to permanent storage but keeps them in memory in a log. Helper function for synchronizing database instances.
- *dbSetOnline(group_path)*: Commits all non-stored metrics from the log and sets the current instance of the database online.
- *getHostNames(group_path)*: List all hosts for which metrics are stored.
- *getLastMetricsByHostName(group_path, host_name)*: Return a MetricSet object that is a list of many Metric objects that are attributed to host_name. Note that the Metric objects do just contain one time-value pair, the most recent one! Other time, value records could be retrieved with getRecordsByMetricName.
- *getLastMetricByMetricName(group_path, host_name, metric_name)*: Retrieve the metric specified by host_name and metric_name. The metric contains the last time and value recorded.
- *getLastSeen(group_path, host_name)*: Return the last_seen timestamp and age for a host.
- *getMetricNames(group_path, host_name)*: List all metric names that are stored for a particular host name.
- *getRecordsByMetricName(group_path, host_name, metric_name, start_s, end_s, nsteps, step_s)*: Return a list that contains record objects. Each record has two attributes time_ns (time in 10E-9 s) and value. The argument start_s in seconds specifies the earliest record to be returned. No records newer than end_s (in seconds) are returned. Finally nsteps defines the number of steps (data points) to return. Like step_s this will lead to averaging for numeric data. The argument step_s gives the minimum time between two consecutive records. This method returns a list containing records.
- *getSummary(group_path, path)*: Returns A "directory" listing of a path inside the fs serialized metric database.
- *getTimeSeriesType(group_path, host_name, metric_name)*: Return the type of time-series stored for a metric on a host, i.e. it's class name. The returned string

contains the class name of the time series for the metric and is either "RRD" or "LOG".

- *findWhereMetric(group_path, metric_name, metric_attr, condition, value)*: Return hosts for which the given metric's attribute fulfills a particular condition. This method is implemented as a fast lookup that only scans the in-memory data and avoids expensive disk operations.
- *hierarchyGroupNames(group_path)*: Helper function that lists hierarchy group paths that are children to the passed group_path parameter. It helps recursing down the hierarchy tree without the need of having explicit knowledge of it.
- *getDBInstances()*: Lists database instances present on this node. Returns a list of group paths for which the current node is a master.

4.3.2 Aggregators

Aggregators are the components of AggMon that probably contribute mostly to its scalability. They are running on group masters and are subscribing to the group's metric channel. Two generic aggregator classes provide the skeleton for the concrete implementations: *Aggregator* and *DPAggregator*. Aggregator is a simple consumer that subscribes to only one topic and gets a channel passed in where to publish its derived metrics. DPAggregator is a dual-ported consumer, it subscribes to the topic of the metric it should aggregate and in addition it subscribes to its own metrics that might get pushed upwards from subgroups.

A set of aggregators were implemented on top of the two generic classes.

Host Aggregators

Host aggregators are actually creating a new, derived metric out of a measured one. The derived metric belongs to the same host as the old metric and is being published with the host's topic into the own group's channel. In that sense they don't actually aggregate data, but transform it. Two host aggregators are currently implemented:

- HostMaxAggregator: an example with little practical use. For a given metric it's largest value since the start of the aggregator is tracked and published. Could be useful, for example, for seeing the maximum swap space used on a node.
- HostSimpleStateAggregator: a complex aggregator that constructs nagios-like state metrics with the values: OK, WARNING, CRITICAL, UNKNOWN out of a measured metric of a host. It allows the definition of states and of conditions that must be fulfilled for the states. Useful, for example, for converting a numeric temperature metric into the more comprehensive states.

Group Aggregators

Group aggregators collect metrics from the own group and transform them into one derived metric on the behalf of the group, which is being published upstream on the hierarchy tree. These metrics very effectively reduce the amount of traffic and data exchanged inside the compute system's management network. At the same time they help finding quickly the problems in the system by descending the hierarchy tree: if the maximum of node temperatures is too large in the cluster this will be reflected by an aggregated metric in the universe group. Finding the exact source of trouble means: look one level deeper, find the subgroup or host belonging directly to the previous level which exceeds the critical temperature. If it is a host, the problem is found. If it is a subgroup, look though its members, recursively. This way only little information needs to be propagated to the root of the monitoring tree, and the detailed information is kept where it belongs to, on the group masters.

The following set of group aggregators have been implemented at the time of writing this paper:

- GroupMaxAggregator, GroupMinAggregator: publish a metric that contains the largest or smallest value of the original metric seen inside the group since the start of the aggregator.
- GroupMaxCycleAggregator, GroupMinCycleAggregator: publish a metric that corresponds to the largest or smallest value of the original metric seen in the latest cycle of measurements. A cycle is the time in which all group members (including subgroups) have published the original metric. In order to avoid waiting forever for lost group members the cycle time has an upper limit after which the aggregated metric is published in any case.
- GroupSumCycleAggregator, GroupAvgCycleAggregator: group cyclic aggregators that publish the sum or the average of the members' metrics seen within a cycle. Subgroup metrics are considered with their weight factor corresponding to the number of members they represent.
- GroupTristateCycleAggregator: aggregates nagios-like state metrics with values OK, WARNING, CRITICAL to a group metric having the value of the worst state seen within a cycle. It can give an immediate overview of the state of an entire group: OK or CRITICAL.

We are currently extending state aggregators to be able to trigger activities through configurable *actors* when states change.

4.4 Commands via RPC

A command channel is needed for different purposes within the monitoring framework. One reason is to send database queries to nodes running a database instance. Another reason is that components need adjustments, e.g., change the data collection interval. Since the publish/subscribe network is asynchronous it does not feature the required functionality. Within AggMon a common Remote Procedure Call (RPC) scenario is used to execute synchronous commands on remote nodes. Commands can also be emitted by a user interface (command line tool or GUI) and represent the data in a decent way.

4.4.1 Data Flow Within the Monitoring Framework

This example describes a system that gathers data from a Ganglia Monitoring System [2], publishes them with a dedicated topic on the Monitoring Framework before they got delivered to a database system to be stored for later retrieval by a command line tool.

The following components are involved in this scenario:

- Ganglia data collector: Ganglia has no mechanism to push data to the collector, thus the collector must actively retrieve data from Ganglia. Hence collection interval can be set by the collector. The collector establishes a connection (TCP/IP in this case) to Ganglia. Ganglia then sends XML formatted data on the socket and closes the connection. The Collector parses the XML data and generates several Metrics. Remember that a single Metric contains only a single measurement value. The final Metrics are published under a particular predefined topic to the AMQP broker.
- AMQP broker: The AMQP broker maintains a message queue each subscriber and topic. If a newly published message topic matches the queue topic of a subscriber the broker adds the message to the queue and invokes the subscribers notify function.
- Database: The database is a subscriber to a particular topic. It stores time-value pairs in a round-robin scheme. Augmented data like unit, source and origin of the data are also stored but overwrite previously delivered data.
- Command line tool: User interaction is a synchronous operation and thus uses the RPC channel to retrieve data from the database. Possible arguments are a particular type of metric or a time-frame within time-value pairs are to be retrieved.

This scenario shows, that the AMQP-based Messaging Framework as a central component decouples data generation at the source, data collection in the database and synchronous data retrieval via the command line tool. Another useful benefit is that developers can easily work on different components together due to the clearly defined interfaces. Usually the mentioned components are implemented as separate processes which increases reliability and adds some degree of fault tolerance (like previously discussed). It is further possible to stop and restart components and to add components during run-time.

5 Conclusion

The aggregating hierarchical monitoring component was built using publish/subscribe messaging and followed the corresponding distributed programming pattern. The inherent decoupling of the program's components as well as the use of Python have speeded up the development allowing for rapid prototyping and enforcing clean and clear interfaces between the components. The publish/subscribe approach is well extensible to other distributed system software for large computer systems, like, e.g., provisioning systems and parallel remote administration tools.

The rather static hierarchy approach for scaling the monitoring workload fits well HPC setups where the structure of the clusters is rather static as well, with dedicated administration nodes. The advantage of the static hierarchy is that compute nodes can be kept free of the heavier monitoring tasks like accumulating metrics, aggregating them and storing or retrieving them from a database. This reduces the potential for monitoring induced OS jitter while naturally enforcing a structure of the monitoring system with groups and group master nodes doing aggregation and holding the pieces of the distributed monitoring database. The contrary approach of P2P or overlay networks based monitoring would apply for rather dynamically managed cloud systems, with nodes being added and removed from the system very frequently. There performance and scalability of parallel programs is rather unimportant, therefore monitoring induced OS jitter can be tolerated.

The decision for using Python and AMQP had a positive impact on the development speed, but later it turned out that AMQP, due to its complexity, limits the message rate to an order of magnitude of 5–10,000 messages per second, while Python's fake multithreading limited the scalability of the daemon that was coded in parallel manner. This limits the performance of a daemon to about 3,500 metrics per second when using a small number of aggregators, but this is more than sufficient to serve the O(100) nodes which we envision to have in a group.

In order to further increase the performance we intend to add a channel implementation that works on top of ZeroMQ [7], a protocol similar to AMQP that has significantly lower overhead. In future we would also like to evaluate NoSQL databases as backend for storing the monitoring data, e.g., MongoDB. Those would allow for a much simpler connection with web frontends and further experiments with data aggregation executed directly on the database, e.g., with map-reduce algorithms.

References

1. AMQP: Advanced message queuing protocol. http://www.amqp.org/ (2012)
2. Ganglia monitoring system. http://ganglia.sourceforge.net/ (2012)
3. Pika. http://pika.github.com/ (2012)
4. py-amqplib. http://code.google.com/p/py-amqplib (2012)
5. Python programming language. http://www.python.org/ (2012)
6. RabbitMQ. http://www.rabbitmq.com/ (2012)
7. ZeroMQ: The intelligent transport layer. http://www.zeromq.org/ (2012)
8. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**, 114–131 (2003). DOI http://doi.acm.org/10.1145/857076.857078. URL http://doi.acm.org/10.1145/857076.857078
9. Marsh, G., Sampat, A.P., Potluri, S., Panda, D.K.: Scaling advanced message queuing protocol (amqp) architecture withbroker federation and infiniband. In: OSU Technical Report. Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210 (2009)

10. van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. Department of Computer Science, Cornell University, Ithaca, NY 14853 (2002)
11. Wang, Q., gang Xu, J., an Wang, H., zhong Dai, G.: Adaptive real-time publish-subscribe messaging for distributed monitoring systems. In: OSU Technical Report. IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, Intelligence Engineering Lab., Institute of Software, Chinese Academy of Sciences P.O.Box 8718, Beijing 100080, China (2003)