Extensible Monitoring with Nagios and Messaging Middleware

Jonathan Reams <jreams@columbia.edu>

CUIT Systems Engineering, Columbia University

Abstract

Monitoring is a core function of systems administration, and is primarily a problem of communication – a good monitoring tool communicates with users about problems, and communicates with hosts and software to take remedial action. The better it communicates, the greater the confidence administrators will have in its view of their environment. Nagios has been a leading open-source monitoring solution for over a decade, but in that time, the way it gets data in and out of its scheduling engine hasn't changed. As applications are written to extend Nagios, each one has to figure out its own way of getting data out of the Nagios core process. This paper explores the use of messaging middleware, in an open-source project called NagMQ, as a way to provide a common interface for Nagios that can be easily utilized by a variety of applications.

1 Introduction

Monitoring is a core function of systems administration, and is primarily a problem of communication – a good monitoring tool communicates with users about problems, and communicates with hosts and software to take remedial action. The better it communicates, the greater the confidence administrators will have in its view of their environment.

Communication is complicated. Marshaling and serializing data is complicated. Ensuring atomicity is complicated. Above all, it is very complicated to do all of this quickly for a large number of connections, but as technology designed to make communication between computers and people easier has advanced, monitoring systems retain the communication methods they were initially designed with a decade ago.

Nagios has been a leading open-source monitoring solution for over a decade, but in that time, the way it gets data in and out of its scheduling engine hasn't changed. As applications are written to extend Nagios, each one has to figure out its own way of getting data out of the Nagios core process. This paper explores the use of messaging middleware, in an open-source project called NagMQ, as a way to provide a common interface for Nagios that can be easily utilized by a variety of applications.

By creating a better interface for moving data into and out of Nagios, we make it easier to solve some of the issues of monitoring very large numbers of hosts and services. In addition to the core of the project, which links Nagios to the outside world, we explore several use cases for NagMQ in a large distributed environment. The project will efficiently distribute the workload of monitoring across dedicated monitoring hosts and to the edge, implement an active-passive failover cluster, and enable easy implementation of alternate user interfaces.

1.1 Background on Nagios

At its core, Nagios is an event loop that runs through a list of checks for a set of services and hosts, executes them, and then executes further commands to notify or take corrective actions when the output of the check changes from the last time it was run. Although some special kinds of checks are run synchronously in the event loop, most are run asynchronously with a special Nagios result collector process that collects the results of the checks and communicates them back to the event loop via the file system. Because each check also has a corresponding collector process, and because Nagios is configured to fork twice per check by default, each check may result in 3 different processes. As checks execute, a small text file is written out by the collector process that contains the textual and numerical output as well as some statistical and timing information for the executed check. The master Nagios process periodically reads all these text files, parses the output of the checks, takes any actions (like notifying the owner of the check or taking corrective action), and schedules the next execution of the check as necessary.

In addition to periodically reading in check results, the master Nagios process also writes out its internal state for use by the user interface. The web interface reads both the configuration files for Nagios and the status data file whenever it needs to display any dynamic information to the user, regardless of how much or what kind of data the user is requesting. These status files can also grow to be very large in large installations of

Nagios – in the production environment described by this paper, the status, cached configuration, and retention data files add up to 54 megabytes that must be read into memory and parsed whenever the user requests some information.

Input into Nagios at runtime is possible through a named FIFO pipe that is opened by Nagios and accessible to the web interface and other addons. Passive check results and commands are formatted into a semicolon delimited line of text and written into the pipe. Separate threads in the Nagios daemon execute any commands coming in from the pipe immediately, and write out passive check results to the file system for later processing by the check result reaper. Because the pipe is just a UNIX FIFO, it has a maximum capacity before clients are blocked (on Linux, the capacity of a FIFO is 64KB, since Linux v.2.6.11).

An optional pluggable event broker (NEB), introduced in Nagios 2, allows developers to register callbacks for most events in the master Nagios process (Galstad, Nagios 2.0b1 Available, 2004). Despite having direct access to the raw data as it moved through Nagios, NEB plugins couldn't override the default execution behavior until Nagios 3.2.2 (Galstad, Nagios Core 3.2.2 Released, 2010). All the Nagios add-on projects mentioned in this paper use a NEB plugin to facilitate communication between Nagios and their application.

The enemy of Nagios is check latency, which is the difference between the actual time a check is executed and the time it was supposed to be executed. Because Nagios is an event loop running in a single thread, latency can rise whenever checks start to fail, and more synchronous host checks need to be run to check dependencies between checks, when checks need to be re-executed to determine whether a problem is transient (to determine the "hard state"), or when notifications need to be sent to a user. As latency goes up, the confidence that Nagios will actually detect problems goes down.

As we shall see, making as many actions as possible in the event loop asynchronous – and thereby reducing the amount of time Nagios spends taking each action – is critical to good performance of the system overall. Design choices – like using the file system, which is backed by disks that are slower than memory, as the transport for check results and for getting data out to front-end interfaces – create challenges to making Nagios scale effectively. To minimize impact on performance, many NEB plugins simply hand off the event to some external daemon to be handled asynchronously.

1.2 Background on Messaging

The purpose of messaging middleware is to provide a framework for the complicated work of communication between software, so that the application can focus on the application, and not moving data. One of the most mature examples of modern messaging middleware is JMS (Java Messaging Services). JMS isn't a single protocol or product; it's an API for Java that supports a variety of messaging patterns. Although there are many products available that support or extend JMS, they are often incompatible with each other. Some examples of JMS-compatible message brokers are Apache's ActiveMQ and Qpid, JBoss Messaging, and RabbitMQ (which also supports AMQP).

When JPMorgan and iMatix developed the Advanced Message Queuing Protocol (AMQP) in 2006, the goal was to create a common open source and standardized protocol for messaging, drawing heavily from JMS. (O'Hara, May 2007) AMQP provided a messaging system that was relatively easy to access via a variety of programming languages and operating systems, and since AMQP is a protocol and standard rather than a product, there are a number of implementations of the broker and client libraries – making it easy to adopt for a variety of applications. RabbitMQ and OpenAMQP are examples of AMQP message brokers.

AMQP, JMS, and other related messaging systems – such as ActiveMQ's support for the STOMP protocol – rely on a central broker that sits between all clients that want to send messages and passes messages between them (iMatix, 2008); the broker is not just a proxy for information, it sits at the center of the application, and a lot of application logic is put into the way messages move through the broker. This design introduces a lot of overhead. Where before there may have been a library that supported communication, now there is a whole infrastructure. The design of Nagios already puts the Nagios daemon at the center of your monitoring with a number of applications and add-ons hanging off of it; adding a new message broker to coordinate communication confuses the picture by putting Nagios core as a leaf instead of a root.

ZeroMQ is a brokerless messaging library written by iMatix – they stopped developing for AMQP and its reference implementation OpenAMQP to work on ZeroMQ (Pieter Hintjens, 2010). Unlike AMQP/ActiveMQ, it does not borrow any semantics from JMS, and acts more like a BSD sockets library – in fact the API makes a lot of effort to be an almost drop-in replacement for standard socket calls. Unlike

JMS libraries, it does not do any data marshaling – message payloads are treated as opaque binary blobs. (Hintjens, Welcome from AMQP, 2012) (Hintjens, ZeroMQ Guide) The one exception is that its subscription socket will do prefix-matching on incoming messages, and discard any messages that do not match any registered subscriptions.

1.3 Related Work

There are a number of projects available for making Nagios scale to large installations and for making it easier to get information into and out of it. We will discuss four different projects that provide similar functionality for Nagios.

1.3.1 DNX

DNX is old enough that the ability to override checks from a NEB plugin in Nagios comes from a patch for DNX. The DNX NEB plugin has a number of internal threads for communicating with clients. When the DNX worker process starts, it connects to the "head node" running Nagios and waits for check jobs. Jobs are executed by the head nodes, and the results are sent back to the DNX NEB for processing. Check results are added to a list internal to DNX that gets merged with the Nagios check result list when the reaper event occurs. (Augustine, 2007) DNX has not had a new release since April 2010, and may no longer be under development.

1.3.2 Mod_gearman

Mod_gearman is another distributed check execution system based on the gearman job distribution framework. In addition to distributing load between different monitoring hosts, the use of gearman allows for more advanced distribution patterns. For example, checks for a remote site can be run at the remote site and results proxied back through gearman, reducing the number of firewall exceptions needed to get the data in and out. Mod_gearman also allows you to duplicate the check results out to a passive Nagios host, providing state replication for high availability. There are also a number of advanced options for selecting which checks get executed where, such as affinity for hostgroups and dedicated workers for hosts, services, and event handlers. (ConSol* Labs) As we shall see in the performance testing, mod_gearman runs check jobs synchronously in its workers – so its performance overall requires that there be a large pool of worker processes and that check duration be very short.

1.3.3 NDOutils

NDOutils is a commonly used add-on to Nagios that is used for getting configuration and state information out of Nagios for insertion into a database or other 3rd party applications. It consists of a NEB module (NDOmod), which serializes Nagios data and transmits it over a socket to files and sockets, and two daemons which process that data into a database; NDO2db inserts configuration sent by NDOmod into a database and LOG2db inserts historical Nagios log information into the database. (Galstad, NDOUtils Documentation, 2007)

1.3.4 Merlin

The Merlin project was started to "create an easy way to set up distributed Nagios installations, allowing Nagios processes to exchange information directly as an alternative to the standard Nagios way using [the Nagios command pipe]." (Ericsson, Merlin) As well as moving data between Nagios processes, Merlin stores its data in a database and serves as the backend of the Ninja frontend project.

Like NagMQ, Merlin is focused on communication with Nagios, with different instances of Nagios taking on different roles – a "NOC", which receives check results and sends configuration data; a "poller", that sends check results to NOC processes; and a "peer", that checks hosts and services redundantly for high availability. (Ericsson, Merlin). It is focused on instances of Nagios communicating with each other for load balancing and fail-over, whereas NagMQ is more general, but in many ways it has the same purpose as NagMQ. The main difference between NagMQ and Merlin, in terms of how they are deployed and used, is that NagMQ depends only on having the ZeroMQ libraries installed – whereas Merlin requires a database and has a daemon running outside of Nagios.

2.1 Architecture

The center of the NagMQ project is an event broker plugin that provides interfaces into Nagios via ZeroMQ message queues. The data from Nagios is serialized into JSON objects/arrays before it is put on the wire – JSON was chosen because it is a relatively well-defined data format with relatively quick parsers for most languages. For the most part, the keys in NagMQ objects correspond to the name of the corresponding value in a Nagios data structure. The three interfaces of NagMQ are:

1. Publisher for events as they happen in the Nagios event loop

2. Command and state update interface

3. State and configuration information query interface

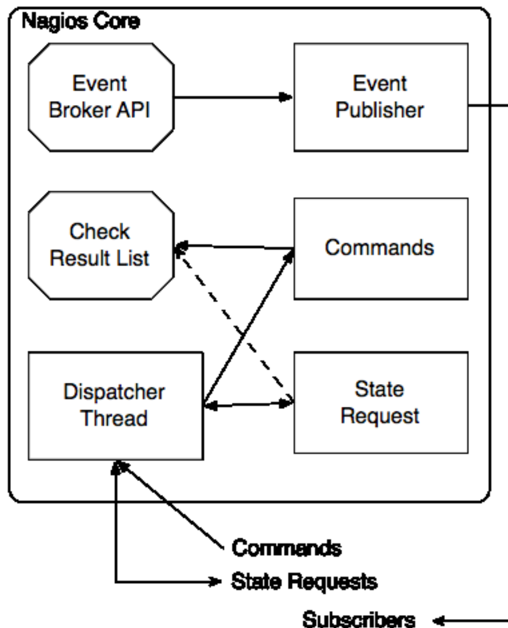


Diagram of the NagMQ NEB internals. The Event Broker API,Check Result List, and Event Publisher all run inside the main Nagios thread

2.1.1 Event Publisher

For the event publisher, messages consist of a prefix that includes the type of message and the relevant host name and service description, if one exists, and the actual message JSON payload. Consumers must subscribe to at least one prefix in order to receive messages. If there are no connected subscribers, the JSON payload is still generated, but is discarded by the message queue.

If a connected client wanted to receive information about all processed service checks for a given host, it would subscribe to the "service_check_processed hostname" prefix – and all other messages would be discarded. In ZeroMQ 3.x, these subscriptions are forwarded to the publisher and only those messages will be sent over the wire, whereas in ZeroMQ 2.x, the filtering is done at the subscriber – this is not an issue except as a consideration of the amount of bandwidth needed to transmit all of Nagios' events. (Sustrik M. , 2011)

Because the actual transmission of messages occurs in a dedicated I/O thread, only generating the JSON actually occurs inside the event-loop, and each call to the broker is essentially asynchronous.

The publisher interface can also be configured to override the default behavior of service checks, event handlers, notifications, and, with a patch, asynchronous host checks, so that they can be executed outside the Nagios daemon by an application subscribing to the message queue – this is used for implementing distributed check execution.

1	`service_check_initiate malanga testexec`
2	`{ "host_name": "malanga", "service_description": "testexec", "check_type": 0, "check_options": 0, "scheduled_check": 1, "reschedule_check": 1, "current_attempt": 1, "max_attempts": 2, "state": 0, "last_state": 0, "last_hard_state": 0, "last_check": 1347985062, "last_state_change": 1343674250, "latency": 0.16800, "timeout": 60, "type": "service_check_initiate", "command_name": "check_by_sr", "command_args": "test!exitcode=0", "command_line": "/opt/local/bin/check_by_sr -H malanga -c test -o \"exitcode=0\"", "has_been_checked": true, "check_interval": 3, "retry_interval": 2, "accept_passive_checks": true, "timestamp": { "tv_sec": 1347985242, "tv_usec": 169135 } }`

Example messages sent by publisher to indicate the beginning of a service check; first column is message part number, second is text as-sent.

2.1.2 Command Interface

Messages sent to the command interface have no prefix and should be a single JSON object containing the type of command or update to be processed, information about the target of the command (e.g. the host name and service description), and any other parameters.

This interface also has its own internal message queues to insert check results into the reaper queue efficiently. The command interface may have multiple worker threads. As check results are received by command worker threads, they are sent to a queue where they accumulate until the reaper event is started and each message is popped off the queue and appended to the list of results to process. This means that check results sent to NagMQ never touch the file system, reducing a potential I/O bottleneck.

In addition to check results, downtimes, comments, acknowledgements, and commands, the command interface will also accept state information for all hosts/services in order to facilitate state synchronization between Nagios hosts at start-up. The interfaces for receiving check results and other state changes expect the same output format as messages published by the event publisher – so the results of one instance of NagMQ can be fed into another for state synchronization.

This is the only interface that does not send any response back to the user, but unlike a publish/subscribe interface, sending a message to this interface will block until the interface is running and able to accept new messages – although messages may be accepted and queued instead of being processed immediately, so successfully sending a message is not a guarantee that it has been processed.

2.1.3 State Request

In addition to listing all hosts, services, contacts, hostgroups, servicegroups, and downtimes, the state request interface also allows clients to query the full state and configuration of individual objects. Like the command interface, the state interface can have multiple worker threads.

By default, the state request interface will return all the information about any objects that match the specification of the query, and clients are encouraged to explicitly list the keys they want returned for their objects. Certain keys (plugin_output, long_output, and perf_data – the texual output of check plugins) require locking the Nagios event loop to ensure thread safety while copying out the data, because they are changed by updated check results. The loop is only locked for the time necessary to copy these three strings out to the payload, so the chance of driving up latency is very low; it is more likely that state requests will take longer if they are issued when the reaper is running.

2.2 Modularity

NagMQ is meant to be as simple or as complicated as necessary for whatever application it is being used. All the interfaces in NagMQ are optional, and NagMQ allows the use of any of the transports and semantics available from ZeroMQ. Endpoints can connect and bind in any order, and clients will transparently reconnect if their peer endpoint becomes temporarily unavailable or if the connect started before the peer had bound to its address. ZeroMQ is brokerless, and clients can connect to and use NagMQ directly without any kind of broker, but the project is distributed with a simple proxy broker for providing advanced routing and for converting one messaging pattern to another (for example subscribing to check initiation messages and fair queuing them to worker processes for distributed check execution).

The NEB module has a dispatcher thread which is enabled when the state request and/or command interfaces are enabled. If they are not configured to run in their own threads, the dispatcher thread calls their processing routines directly. Otherwise, it dispatches commands and requests from connecting clients to the dedicated worker threads.

2.3 Security

NagMQ does not provide any encryption or authorization for messages, and ZeroMQ does not provide privacy or verification for messages. ZeroMQ 3.x has an option to restrict TCP connections to certain address ranges, but ZeroMQ 2.x, connection-level security must be done externally (e.g. through a firewall, or IPSec, etc.). Future work may include adding encryption to NagMQ by running JSON payloads through a block-cipher with a pre-shared key before being put on the wire.

2.4 JSON

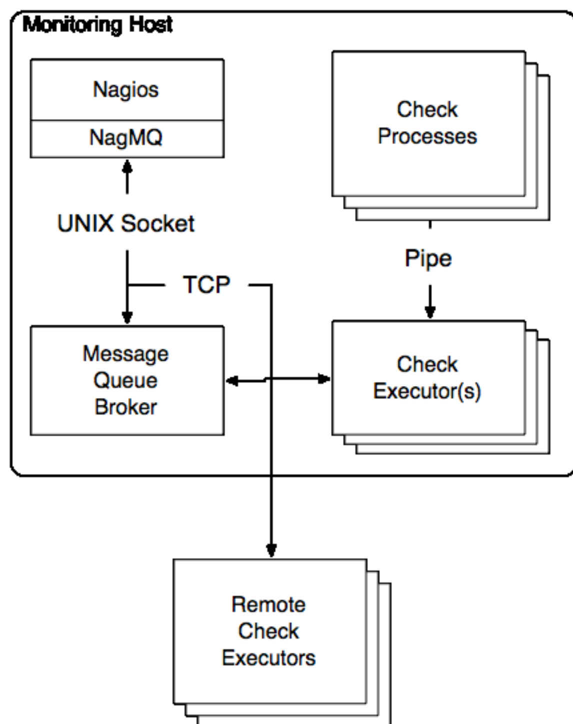
Since NagMQ sends and receives data as JSON payloads, the speed of the parser and emitter has as much of an impact on the performance of the whole module as the messaging system the data is transmitted over. NagMQ has its own write-only JSON emitter, which is used in the event publisher and state request module. Its performance is similar to the performance of the standard C library sprintf – in fact, most calls are just wrappers to sprintf and other standard C calls, with some additional memory handling. Any values needed for constructing a message prefix are cached as they are added, and the resulting buffer is freed by ZeroMQ when it has finished being transmitted to peers.

The command and state request modules also use the Jansson JSON library for parsing requests from clients. Although it is slower than the JSON emitter, because it constructs a hash table for each object it parses, all the modules that use it can be run in multiple separate threads to ensure high performance.

3. Included Applications

Included with the NagMQ event broker are a ZeroMQ proxy and a number of scripts and applications intended to show the possible uses of NagMQ; unlike many other Nagios add-on projects, NagMQ is not intended to be a single add-on that solves every problem on its own, so the included applications are intended to be examples that can be built upon when deploying Nagios, and many of them are written in Python and include no compiled code. We will discuss distributed check execution, high availability with active/passive failover, and some alternate user interfaces that all use NagMQ for communicating back to Nagios.

3.1 Distributed Check Execution



NagMQ includes a distributed check executor that subscribes to messages for the start of a service/host check and executes the command line of the check on whatever host the executor is running on. The executor itself is entirely asynchronous, using LibEv to coordinate I/O with child processes, reap return codes from exited children, and receive new events from ZeroMQ. When connecting to a message broker instead of subscribing directly to the event publisher, jobs will be automatically fair-queued to connected peers by ZeroMQ, and adding capacity requires only starting additional executor processes. The executor can filter which checks it runs either by changing the subscription on the message bus, or by applying a Boolean match against any field in the check initiation payload.

In addition to distributing checks across dedicated monitoring hosts, the executor can also be deployed to the edge as a remote plugin executor. Simply by subscribing to the beginning of service checks for a specific host and applying some clever check naming, the execution agent will run all remote checks as they happen in the Nagios event loop and send the results back without Nagios ever having to start a new process or run a command. At the edge, the remote-execution agent can be completely stateless, with the configuration only telling it how to connect to the message queue, rather than having to list explicitly each command the monitoring system is allowed to run.

3.2 Active-Passive Failover

A basic agent that provides active-passive high availability for Nagios comes with NagMQ and uses a periodic program status message as a heartbeat to detect when the active node has failed. When the agent starts, it launches Nagios with all active checks, notifications, and event handlers disabled. It then requests the state of all the hosts and a service configured on the active node from the state request interface and submits the result to the newly started passive node's command module – it has a special routine for parsing the current state of an array of hosts and services for this purpose. It resolves differences between downtime and comments, performs any fencing required to ensure the active node is not disrupted, and enters a loop of receiving events for check results, downtimes, comments, and acknowledgements and forwards them into the passive node.

If there is a timeout in receiving a check result or other state message to be forwarded to the passive node or a program status heartbeat message, the agent will perform any fencing necessary to become the active node, and send the necessary commands to the node being promoted to start active checks, notifications, and event handlers.

Once a node has become the active node, it receives and discards its own program status message and will perform fencing actions to become passive if it detects Nagios has failed. This gives reasonable assurance that if Nagios on the active node has failed but the HA agent/OS is still running, any shared resources like IP addresses can be cleanly taken over by the new active node.

3.3 User Interfaces

Stock Nagios comes with a web interface that uses a number of CGIs to parse Nagios status data and submit

commands back through the command pipe. NagMQ allows for multiple user interfaces that are able to obtain their data from the state request interface of NagMQ and submit commands through the command interface. In addition to being able to query only the information that the user needs to display, the different transports available to NagMQ allow the user interface to be on another host, or a group of hosts, that are separate from the Nagios monitoring hosts.

NagMQ comes with a command-line interface for Nagios which allows users to view current status – add and remove acknowledgements, downtime, and comments – and enable and disable checks and notifications. It uses the username of the user invoking the command as the contact name to determine whether a user is authorized to submit commands for given targets, and can act on hosts, hostgroups, and service names.

4 Performance

Testing the performance of NagMQ is difficult because it is not a single-purpose application. While developing the applications included with NagMQ, it was often the case that performance problems occurred when the program connecting the message queue couldn't decode the JSON payload fast enough – this is why the distributed check executor is implemented in C.

Since check execution uses several parts of NagMQ (the publisher to receive check events, the message queue broker to divide work up between connected peers, and the command interface for receiving check results), we will compare the check latency and the number of checks run by the check executor per minute to demonstrate the relative performance of NagMQ to stock Nagios and an existing distributed check executor.

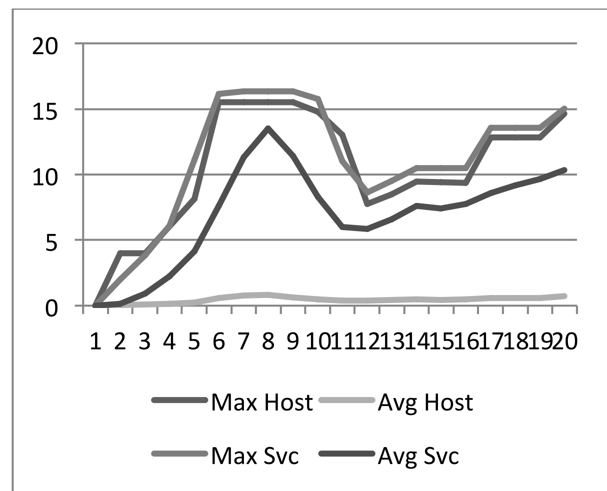
We tested Nagios 3.4.1 in its stock configuration, with NagMQ and its included distributed check executor, and Mod_gearman. Nagios was configured with 20,000 service checks across 2,000 hosts with a dummy check script. The check interval for services was every 3 minutes with a 1-minute retry interval for services – services were configured for 2 check attempts to determine a hard state; and hosts only checked once. Each test ran for 20 minutes, and Nagios started each time with no state from any previous tests.

The check script printed a random string, slept a random number of seconds, and returned a semi-random return code. The return code was based on the random number used for selecting the string that was printed, but was weighted towards returning a successful return code 90% of the time, and a problem 10% of the time. The checks routinely returned an error in order to test the code paths in Nagios that can introduce latency, such as on-demand host checks, retries, and executing many notifications.

All performance graphs have time in minutes on the horizontal axis, and latency in seconds on the vertical axis. Averages for checks executed per minute exclude the first sample for all tests, because Nagios had not been running for a full minute when they were sampled.

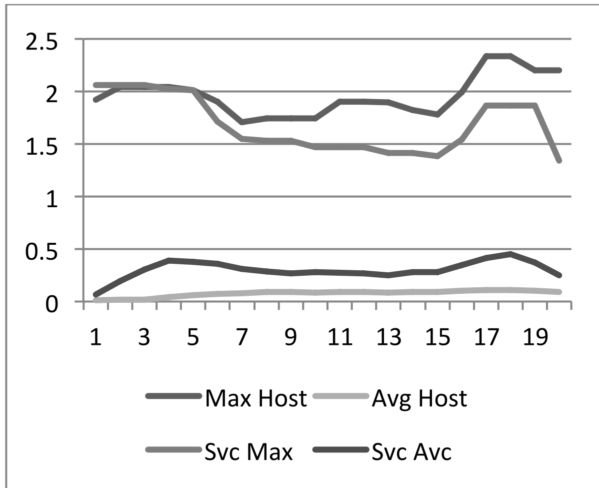
4.1 Stock Nagios

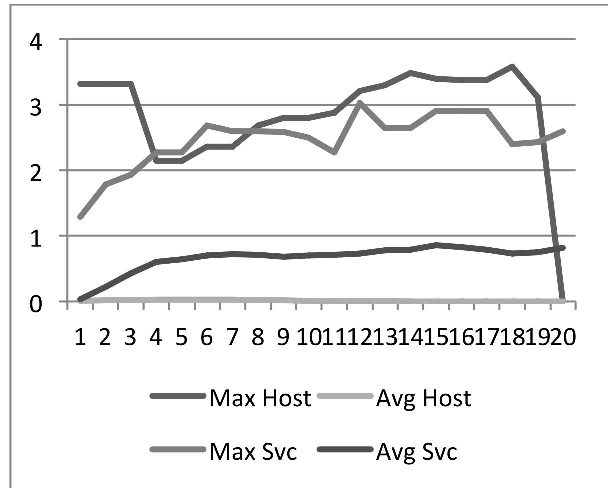
Check latency increased gradually over the course of the test with stock Nagios, with a maximum latency of 14.646 seconds for hosts and 14.014 seconds for services, and a maximum average of 0.73 seconds for hosts and 10.35 for services. Nagios was able to consistently execute an average of 976 host checks and 6219 service checks per minute.



4.2 NagMQ

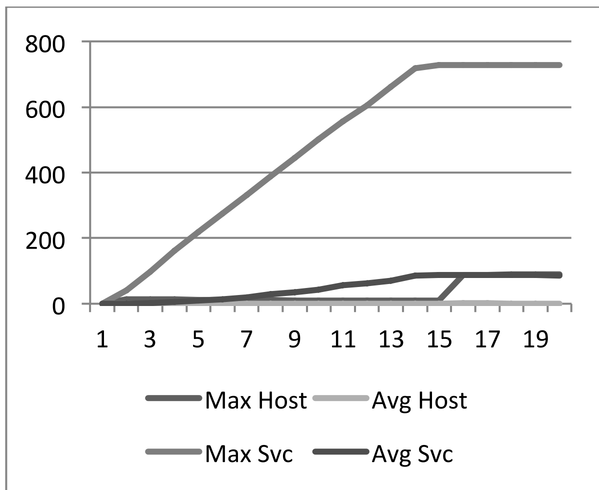
The average check latency for both hosts and services never went above 0.5 seconds during the test, and the maximum check latency was above 2.5 seconds. The executors consistently ran an average of 1,028 host checks and 6,726 service checks per minute.





4.3 Mod_gearman

Mod_gearman had the least expected performance characteristics of the three environments. The first test was run with the check script sleeping a random number of seconds, and the average check latency went up consistently. The number of checks executed per minute was an average of 828 host checks and 747 service checks.



Because performance was so poor, the check script was modified so it no longer slept, but returned immediately. The performance in this test was much better, with check latency for hosts dropping to 0 by the end of the test. The workers consistently executed 1,507 host checks and 7,063 service checks per minute.

4.4 Results

The performance of NagMQ overall varies for different applications, but testing of the distributed executor shows similar performance to popular distributed executors like mod_gearman, with latency below 1 second. Performance of the NagMQ event broker is broken down into two categories: how quickly it can generate JSON message payloads and how quickly it can send and receive payloads.

The JSON emitter that comes with NagMQ was tested by generating 20,000 test payloads with constant data similar to the results of a service check – the equivalent of 16.9 MB of check data – and took 0.27 seconds.

Testing of NagMQ was performed on virtual machine configured with 4 CPUs and 8GB of memory. ZeroMQ is designed to move data around very quickly, and although results will vary with different hardware, the performance of ZeroMQ on our test platform is shown below:

100,000 roundtrips of 4096 bytes via UNIX sockets	
Latency	43.497 [us]
Throughput	138432 [msg/s] 4536.140 [Mb/s]
100,000 roundtrips of 4096 bytes via TCP/IP	
Latency	63.169 [us]
Throughput	87109 [msg/s] 2854.388 [Mb/s]

Problems inside Nagios remain that limit the performance of the system overall. Recent builds of Nagios contains many fixes and improvements that make it faster overall, such as replacing the scheduling queue doubly-linked list with a priority queue (Ericsson, Commit #1971, 2012). In older versions of Nagios, any performance gains from using NagMQ may only be covering up an underlying inefficiency.

5 Future Work

The NagMQ project is mature enough for production use, but there are a lot of enhancements possible to expand its usefulness and improve monitoring with Nagios. The development of NagMQ has been very organic thus far. The first version was intended to include only the event publisher, with applications either waiting for events to determine state or submit commands via the command pipe. The addition of the command module was introduced next to implement distributed check execution. Finally, the state request module was added to implement a command-line interface. There have been additions to all modules as new applications were developed.

The design of NagMQ is limited to a single messaging system and a single payload format, and there have been requests from users for support for other messaging systems, which would also mean different data formats. Although the input portion of NagMQ would have to be rewritten, the emitter could be adapted to support different data formats. Future work could support modular support for different data formats and messaging systems.

NagMQ has no security that is not provided by ZeroMQ, and is limited to ACLs for restricting what IP addresses and CIDR ranges are allowed to connect to a bound address. Future versions of NagMQ and its included applications should include improved security both for authenticating to NagMQ, and for providing encryption for all traffic in and out of the NEB.

Presently NagMQ is only able to receive state information from clients – the configuration of Nagios is still firmly rooted in text files. Adding the necessary interfaces to NagMQ to allow it to receive configuration items and add them to Nagios at run-time would allow Nagios to become completely stateless.

Although it had not been released at the time this paper was written, Nagios 4.x was in testing, and the way Nagios executes checks and communicates with itself has changed significantly in the test builds of Nagios 4. Instead of running checks (Ericsson, Commit #2020, 2012), notifications (Ericsson, Commit #2026, 2012), and processing incoming events in the event loop, it uses dedicated worker processes and asynchronous message passing to get execution out of the event loop. This change focuses Nagios core on being a better scheduling engine, and moves its design closer to the design of NagMQ.

6 Conclusion

The popularity of Nagios leads to a hesitancy in changing the way it interacts with the checks it needs to run, users, and other applications. Although there are a number of different projects that improve specific parts of communication with Nagios, NagMQ attempts to provide a generic interface that is easily accessible from scripts and applications. Although the performance of communicating with NagMQ can vary widely depending on the performance of the JSON parser and emitter used, in the best case it has performance comparable to other popular projects for low-latency applications like check execution.

A big focus of NagMQ was that it should have minimal dependencies and not require many patches to stock Nagios in order to be useful. There were changes required to Nagios to ensure communication between the state request interface and the event loop is thread-safe, which have already been submitted back to the Nagios project. Otherwise, the project includes the Jansson JSON parser and libev, which are used in both the NEB module and the distributed executor, but requires the user to provide the ZeroMQ libraries for C and any scripting languages they wish to use.

NagMQ is licensed under the Apache 2 license and available online at https://github.com/jbreams/nagmq.

7 Works Cited

Augustine, A. (2007, October 17). *DNX Version 0.13 Released!* Retrieved September 13, 2012, from DNX Announce Mailing List: http://sourceforge.net/mailarchive/message.php?msg_id=89683

ConSol* Labs. (n.d.). *Mod_gearman*. Retrieved September 13, 2012, from http://labs.consol.de/nagios/mod-gearman/

Ericsson, A. (2012, June 21). *Commit #1971*. Retrieved from Nagios Subversion Repository:

http://nagios.svn.sourceforge.net/viewvc/nagios?view=revision&revision=1971

Ericsson, A. (2012, July 9). *Commit #2020*. Retrieved from Nagios Subversion Repository: http://nagios.svn.sourceforge.net/viewvc/nagios?view=revision&revision=2020

Ericsson, A. (2012, August 2). *Commit #2026*. Retrieved from Nagios Subversion Repository: http://nagios.svn.sourceforge.net/viewvc/nagios?view=revision&revision=2026

Ericsson, A. (n.d.). *Merlin*. Retrieved September 13, 2012, from Op5 Community Exchange: http://www.op5.org/community/plugin-inventory/op5-projects/merlin

Galstad, E. (2004, December 15). *Nagios 2.0b1 Available*. Retrieved September 13, 2012, from Nagios-announce mailing list: http://sourceforge.net/mailarchive/message.php?msg_id=210063

Galstad, E. (2007, April 18). *NDOUtils Documentation*. Retrieved from Nagios Core Manuals: http://nagios.sourceforge.net/docs/ndoutils/NDOUtils.pdf

Galstad, E. (2010, September 1). *Nagios Core 3.2.2 Released*. Retrieved September 13, 2012, from Nagios Announce Mailing List: http://sourceforge.net/mailarchive/message.php?msg_id=26080524

Hintjens, P. (n.d.). Retrieved from ZeroMQ Guide: http://zguide.zeromq.org/page:all

Hintjens, P. (2005). *Background to AMQP*. Retrieved September 13, 2012, from ZeroMQ Wiki: http://www.zeromq.org/whitepapers:amqp-analysis

Hintjens, P. (2012, September 9). *Welcome from AMQP*. Retrieved from ZeroMQ Wiki: http://www.zeromq.org/docs:welcome-from-amqp

iMatix. (2008, June 10). *Introduction to OpenAMQP*. Retrieved September 13, 2012, from OpenAMQP: http://www.openamq.org/doc:user-1-introduction

O'Hara, J. (May 2007). Toward A Commodity Enterprise Middleware. *ACM Queue*, 48-55.

Pieter Hintjens, M. S. (2010, April 23). *Multithreading Magic*. Retrieved September 19, 2012, from ZeroMQ Wiki: http://www.zeromq.org/whitepapers:multithreading-magic

Sustrik, M. (2008, December 12). *Broker vs. Brokerless*. Retrieved from ZeroMQ Wiki: http://www.zeromq.org/whitepapers:brokerless

Sustrik, M. (2011, September 2). *0MQ/3.0 pubsub*. Retrieved from ZeroMQ Wiki: http://www.zeromq.org/whitepapers:0mq-3-0-pubsub