Improving Resiliency and Scalability

in the Real Traffic Grabber

By

Justin Zipkin

A Project Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: INFORMATION TECHNOLOGY

Approved by:

_____
Eugene Eberbach

Rensselaer Polytechnic Institute
Hartford, CT

April 2014

Table of Contents

List of Tables

List of Figures

Abstract

This project considers the state of the Real Traffic Grabber (RTG), a well-known, high performance SNMP statistics monitoring system. First, the current functionality of RTG is analyzed by deconstructing the package into its three core components: rtgpoll, rtgplot and rtgtargmkr.pl. Building upon this analysis, certain scaling problems with large deployments are highlighted, along with current remediation techniques. Next, recent innovations in distributed computing are taken into consideration, and a newly designed architecture is presented where scalability is a key factor. Specifically, a distributed system utilizing ZeroMQ for concurrency and message queuing is proposed to provide ease of configuration, increased fault tolerance, and rapid deployment. Finally, a proof of concept implementation of the newly proposed architecture is provided as a building block toward a production ready, distributed implementation of RTG.

Introduction

As the Internet continues to expand its reach, communication networks grow in size and complexity to meet ever growing technical and political demands. Network operators have more tools today for assisting them than has ever been the case in the past. Despite this, many tools whose development has been stagnant for several years are still prevalent in networks around the world, and these old tools are even being deployed in new installations.

This paper focuses on a piece of software known as the Real Traffic Grabber (RTG), a "flexible, scalable, high-performance SNMP statistics monitoring system" (Beverly, RTG: Real Traffic Grabber, 2007). Although there are many options for network operators who would like to monitor SNMP statistics, RTG makes it particularly easy to collect simple counter values for a high number of devices in customized, short polling intervals. This project looks at the development of RTG over the years, and considers the current state of the software given the patches, forks, and other ad hoc changes that have been made over time. Additionally, several shortcomings of the software are presented which focus around the ability of RTG to scale in a growing environment. The core reasons for these shortcomings are analyzed, and current day remediation techniques are presented.

After the current state of RTG has been covered, a bottom-up approach is considered for a complete redesign that may be a better solution to scalability issues than the techniques utilized today. Modern distributed computing techniques are considered when proposing a redesigned version of RTG, but backwards compatibility is also held paramount to ensure the seamless integration of previously collected RTG data, and tools for working with that data. After presenting this alternative design, a proof of concept implementation of the new architecture is provided as a building block for a production-worthy distributed RTG polling system.

About RTG

RTG is the Real Traffic Grabber, a system designed by Robert Beverly throughout the early 2000's and subsequently enhanced by various contributors to the open source community. In order to clearly present the challenges involved in deploying and enhancing an instance of RTG it is important to understand the different pieces of the system that work together in a standard installation. Although pieces can be swapped in or out, or not used at all, all of the major components have been designed to work in tandem with each other. The following programs make up the collective that is most commonly referred to as RTG.

rtgpoll

At the heart of the RTG system is the polling daemon: **rtgpoll**. It is in charge of polling network devices and relaying information gathered back to a database. The program is written in C and designed to be fast and efficient. It is fully multi-threaded and contains logic intended to speed up the gathering of data, and prevent erroneous data from being inserted into the database.

rtgtargmkr.pl

Managing the configuration file for **rtgpoll** by hand can be somewhat cumbersome. As a result, **rtgtargmkr.pl** was designed to automate the generation of configuration lines. It works by accepting a simple list of network devices and configuration options, and then walking each device's SNMP object tree to create a valid configuration file for **rtgpoll**.

rtgplot

In order to visually analyze data that has been collected with **rtgpoll**, a C program utilizing the GD library is provided. This allows images to be generated on the fly and optionally embedded into web pages. Several parameters are accepted by **rtgplot** in order to customize the output of the visualization.

Other dependencies

RTG as a whole does not rely on many pieces of external software, although it does make extensive use of a few large packages. For storing the polled values, a database is required. By default, everything has been built for MySQL and that is what the official RTG documentation recommends using. PostgresSQL and Oracle database drivers are also included and there are reports of people successfully using these drivers with certain versions of RTG.

In order to query SNMP values off of devices, RTG is also dependent on an SNMP library. By default this is Net SNMP, and it is highly recommended that users compile RTG with Net SNMP available. Alternatively, certain versions of RTG are able to utilize the older UCD SNMP packages, but this is not recommended. In order to use **rtgtargmkr.pl** users will also require a Perl interpreter with DBI. Finally, certain reports that come with RTG will require PHP and a web server.

History of RTG

Like many community-sponsored pieces of open source software, there is no real release cycle for RTG. One can track the general life cycle of the software and its derivatives by

Version	Released	Source
RTG 0.6	2002-07-23	Sourceforge (removed)
RTG 0.7	2002-08-07	Sourceforge (removed)
RTG 0.7.1	2002-08-28	Sourceforge Package
RTG 0.7.2	2002-10-04	Sourceforge Package
RTG 0.7.3	2003-02-08	Sourceforge Package
RTG 0.7.4	2003-10-02	Sourceforge Package
RTG 0.7.5	2003-11-03	Sourceforge CVS
YRTG (patch)	2004-08-30[1]	Yahoo (RTG mailinglist)
RTG 0.8	2005-02-26	Sourceforge CVS
RTG 0.9	2008-01-18	Sourceforge CVS
JRTG	2008-04-01	JRTG Sourceforge
RTG2 (0.9.1)	2012-02-17	Google Code

Table 1 – RTG release dates

[1] Date preliminary code was released on the official RTG mailing list. The current YRTG patch was generated off the RTG CVS code during 2005.

scouring mailing lists and the commit history of various revision control systems. In doing so, it becomes apparent that many different people have identified identical problems with RTG and different fixes have been pushed out in various mediums. Table 1 above shows a history of the most common versions of RTG currently being run in the wild and their respective release dates.

One of the most notable issues that can be seen in the table is the distribution of YRTG, a patch released by Bill Fumerola during his tenure with Yahoo. YRTG addressed many concerns raised on the official RTG mailing list and had several ad hoc improvements. Originally, pieces of the patch were sent out via the RTG mailing list, and eventually a single patch was put up on the web for people to download. Unfortunately, the patch only built against CVS during the RTG 0.8 timeframe, and many of the enhancements never made it into the subsequent RTG builds that continued to be worked on by other individuals in the community. As of RTG2, several internal components of RTG have been redesigned that address limitations within RTG, but make it entirely impossible to apply the YRTG patch.

Further adding to the confusion, if one is to search the web for the latest version of RTG, they will likely find themselves on Sourceforge viewing the official RTG project. The latest packaged download available is RTG 0.7.4, which represents the state of the RTG project in October of 2003. Development continued on the software (in the Sourceforge CVS repository) for years afterwards, and packaged versions of RTG2 can be found on a separate Google Code repository that has been updated around 2012.

Configuring RTG

There are 2 configuration files that are crucial to the operation of RTG. First is the **rtg.conf** file, which is read into memory when **rtgpoll** begins. This file contains basic configuration parameters including: default SNMP information, database credentials, the number of threads to fork, poll intervals and out of range parameters.

In early versions of RTG, a variable called *OutOfRange* was used for globally limiting what a sane counter value was defined as and to prevent bad data from being inserted into the database. In more recent versions this has been replaced with the speed variable as defined below.

The second configuration file is where all polling targets are defined: **targets.cfg**. In order to simplify the creation of the **targets.cfg** file, RTG provides the **rtgtargmkr.pl** script which, when run against a list of routers, will output a valid **targets.cfg** file. While the **targets.cfg** file syntax changed substantially around version 0.8 of RTG, it still contained similar information. In a new installation of RTG, a very basic target file may resemble the configuration lines shown in Figure 1 below.

```
host 10.10.0.1 {
        target .1.3.6.1.2.1.31.1.1.1.7.581 {
                bits 64;
                table ifInUcastPkts_1;
                id 101;
                speed 1800000000000;
                descr "40ge to site2";
        };
        target .1.3.6.1.2.1.31.1.1.1.6.581 {
                bits 64;
                table ifInOctets_1;
                id 101;
                speed 1800000000000;
                descr "40ge to site2";
        };
        target .1.3.6.1.2.1.31.1.1.1.10.581 {
                bits 64;
                table ifOutOctets_1;
                id 101;
                speed 1800000000000;
                descr "40ge to site2";
        };
        target .1.3.6.1.2.1.31.1.1.1.11.581 {
                bits 64;
                table ifOutUcastPkts_1;
                id 101;
                speed 1800000000000;
                descr "40ge to site2";
        };
};
```

Figure 1 – Example targets.cfg file

The targets file is defined as a series of hosts, where each host has a series of targets. A target is a unique SNMP OID that should be queried on the host that it falls under. In the example shown to the left, four separate OIDs are being polled for a single interface. The OIDs in this case map to standard MIBs: ifHCInUcastPkts, ifHCInOctets, ifHCOutOctets, and ifHCOutUcastPkts respectively. In a real world deployment of RTG one would expect to be polling several OIDs for many

interfaces on even more hosts. Thus, the configuration would grow to be many times larger than the simple example presented here.

In addition to the **targets.cfg** file simply defining the target OID values to poll, a few parameters are associated with each of the targets.

bits *– The size of the SNMP counter in bits (32 or 64)*
table *– The name of the database table that results should be written to*
id *– The unique ID for the interface to be recorded in the table*
speed *– The maximum counter delta one would expect to see recorded during a poll interval*
descr *– Some informative text describing what this OID represents*

Most of these fields are self explanatory, with the exception of *speed*. This was implemented in later versions of RTG to replace the global *OutOfRange* configuration parameter. By default, **rtgtargmkr.pl** uses the following calculation to determine the appropriate speed value for a target:

$$s = \frac{i \times p \times c}{8}$$

Figure 2 – rtgtargmkr.pl speed calculation

where :

s = speed value
i = the interface speed in bits per second
p = the polling interval in seconds
c = a "fudge factor" constant, defaulted to 1.2

The speed value is used by **rtgpoll** as a safety check to ensure that the difference between the previously polled value and the most recently polled value is sane, given the time that has elapsed between the two polls. The rationale for this configuration parameter will become clearer after the general lifecycle of a poll is explained.

Anatomy of a poll

RTG is capable of monitoring both gauge and counter values. When monitoring a gauge value, rtgpoll will simply relay the value polled back to the database. When monitoring a counter value, RTG will do the appropriate math to calculate a delta between the last two consecutive polls. This means the first poll that RTG makes is useful only to populate the internal RTG data structure and to mark a date as a placeholder in the database. Following database insertions by the same poller will then contain the difference in the counter value as described below. The logic here can be slightly complicated by a couple of factors, which RTG attempts to mitigate with some wisely chosen configuration variables. For reference during this discussion, see Figure 3 to observe the lifecycle of a poll as it pertains to counter values.

Due to the fact that SNMP counters are only represented as either 32-bit or 64-bit integer values, most network devices will simply wrap the counter back to its starting value when the upper bound is reached. RTG handles this by noticing if the most recent counter value is less than the value that was read in the prior poll. If it is, then it will calculate the delta between the two counter values by taking into

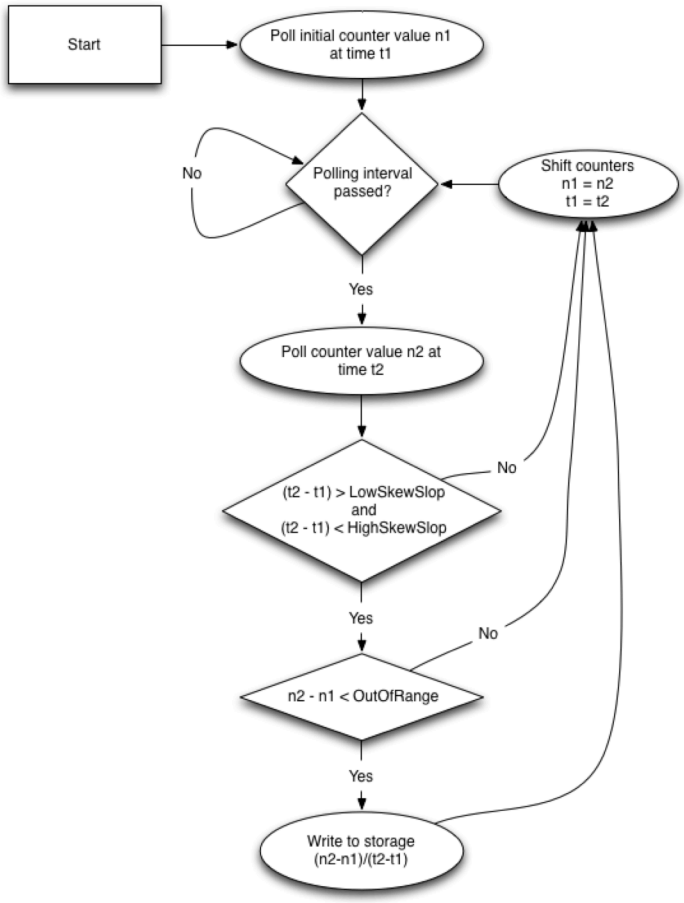

Figure 3 – Poller lifecycle

account the distance between the most recent counter value and the upper bound of the counter (2^{32} or 2^{64}). Otherwise, the delta will simply be the value of the most recent poll minus the value of the poll before that.

Unfortunately, there are other circumstances that may cause a counter value to prematurely reset before it has reached its upper bound and, due partially to SNMP being a stateless protocol, there is often no definitive way for the poller to recognize these. The most obvious case is when a device is reset, but some software also allows a user to manually reset the SNMP counter statistics by issuing a command. To mitigate the effects of this, RTG exposes three additional global configuration parameters: *HighSkewSlop*, *LowSkewSlop*, and *OutOfRange* (replaced by *speed* as described above). By tuning these values to ones network environment, a network operator can be relatively certain that the delta values being generated by RTG are accurate.

HighSkewSlop and *LowSkewSlop* are used to define the maximum and minimum numbers of poll intervals allowed between two consecutive poll values before the time is too large or too small to calculate a valid rate. During the time period when 64-bit counters were scarcely implemented, and network links were starting to increase in speed, these configuration parameters were extremely important. To understand exactly where a *HighSkewSlop* or *LowSkewSlop* corner case may exist, consider the scenario of a 100Mbps interface operating at 90% capacity being monitored with a 32-bit counter. Given the above information, we can expect the counter to wrap approximately every *t* seconds in the equation shown below.

$$t = \frac{2^b}{u \times s} = \frac{2^{32}}{0.9 \times 10^8/8} \cong 382$$

Figure 4 – Expected counter wrap interval (32-bit)

Being that a typical RTG deployment will poll interfaces in 5 minute (300 second) intervals, a *HighSkewSlop* of 2 would be imperative for an accurate delta calculation. In the event that a counter wraps twice, RTG has no way of calculating the actual difference between the two previous counter values.

Whereas the example presented in Figure 4 is effective when we expect frequent wraps, it does not help in a situation where a counter has unexpectedly wrapped. It is in this scenario that the *OutOfRange* and *speed* configuration parameters become important. *OutOfRange* is used to define an upper bound of which RTG will not insert a value into a database (Beverly, rtgpoll, 2003). This is an additional safety check that is layered on top of the concept of a counter wrapping, although it is not foolproof. To reuse the example above, now consider the extreme case of the same interface being polled on a 64-bit counter value.

$$t = \frac{2^b}{u \times s} = \frac{2^{64}}{0.9 \times 10^8/8} \cong 1.64 \times 10^{12}$$

Figure 5 – Expected counter wrap interval (64-bit)

In this case, a counter wrap is not expected for thousands of years, so there is no chance of multiple wraps happening between two consecutive polling intervals. There is, however, the chance that the counter will be reset due to a device reboot or some other event. Before inserting any value into the database, RTG will check the *OutOfRange*/*Speed* value, and if the delta it has calculated exceeds that value, it will instead throw that value away and restart its polling cycle. This prevents unrealistic spikes from showing up in the database and works in most situations as long as the *OutOfRange*/*Speed* value is appropriately tuned. The only exception to this is in the event where a counter is reset when the current counter value is near the maximum counter value (within a factor of *OutOfRage*/*Speed*). This is highly unlikely for 64-bit counters, but is a potential area of concern for 32-bit counters.

The RTG database schema

RTG ships with a default database schema that is largely unchanged in any of the available upgrades. From the early days of RTG, users have been encouraged to modify the database schema to their particular needs, and as such it would appear that contributors to the RTG project have mostly neglected to push out improvements in this area. Additionally, all available versions of RTG discussed in

this paper are intended to be backwards compatible with database schemas that may have been used with previous versions of RTG. The default schema can be seen in the entity relationship diagram pictured in Figure 6 below.

A few points are worth noting about the default RTG schema. First, the foreign keys being displayed do not actually exist in the default schema. RTG was originally made in the early 2000's and the default DBMS was MySQL. MySQL did not include foreign key checking until version 3.23.44 was released on October 31, 2001 (MySQL, 2001). Even at that point, foreign keys were only offered in the InnoDB storage engine, which was not used by default and had nowhere near the following of MyISAM at the time. While things have since changed in terms of the default MySQL storage engine, the current schema released with RTG looks as it did in the initial releases.

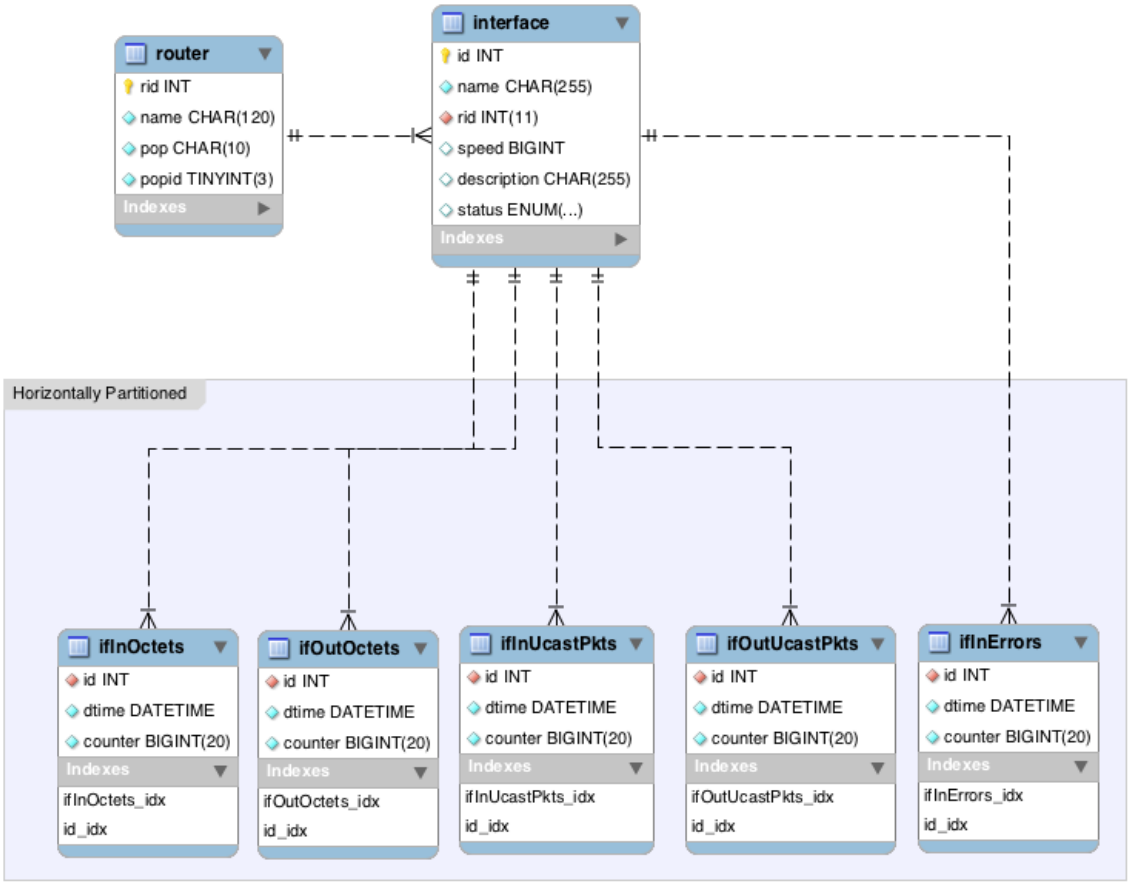

Figure 6 – EER of default database

Another important note about the default RTG schema is the partitioning scheme. When RTG is first configured, a script called **createdb** is run to create the tables as pictured in the EER diagram. By default, **rtgpoll** and **rtgtargmkr.pl** utilize a horizontal partitioning scheme for segmenting out data by router. Whenever a new device is added to the *router* table, **rtgtargmkr.pl** also creates duplicates of the octet, packet, and error tables by appending the router id. For example, when the first router is created, new tables are created that look like: *ifInOctets_1*, *ifOutOctets_1*, *ifInUcastPkts_1*, *ifOutUcastPkts_1*, and *ifInErrors_1*. This allows the table name to be used as an implicit key referencing the router id, and also keeps tables sizes somewhat more manageable in the long run.

It is important to note that there are some key deficiencies with the default database schema included with RTG. Although it is not apparent on the EER diagram included here, there are inconsistencies with the default data types provided by **createdb**. The router and interface IDs are inconsistently implemented as signed vs. unsigned integers, and the *counter* field is only implemented as signed, even though it will never actually be less than zero. Additionally, RTG2 implements a new field on the octet and packet tables: *rate*. In the RTG2 implementation *rate* is also defined as a signed integer, where it should not be. In addition to data type problems, there are some issues with the default indices. The most obvious of which can be seen when **rtgplot** attempts to query based on both *dtime* and *id* in the same query. This causes a full table scan on every attempt, which can be detrimental to performance of a very large table. This can be simply fixed by creating a multi-column index on both of these columns.

Problems with RTG

As with any piece of software, time has exposed many small bugs with RTG. Members of the open source community have fixed some of these bugs, and some

remain outstanding to different degrees (as seen on the RTG2 issue tracker[2]). Rather than focusing on these smaller issues individually, this project looks at problems caused by the particular way in which rtgpoll has been architected.

Firstly, consider the model whereby polls are executed. The main RTG thread is in charge of walking the hash of targets in memory and then passing individual query requests off to workers. From there, each individual poll as described in the Anatomy of a poll section essentially operates within its own dedicated thread. This means that threads will each get their own connection to the database for storing results, and each thread will issue a single SNMP_MSG_GET request, block, and decide what to do with the response on its own. Given this information, it is possible to model how long an entire poll cycle might take to execute. First, consider that for each thread, the run time can be represented as the following function:

$$t(n) = \begin{cases} t_{oh} + p_{max}, & t_{poll} \geq p_{max} \\ t_{oh} + t_{poll} + t_{db}, & t_{poll} < p_{max} \end{cases}$$

Figure 7 – Time to complete an individual poll

$t\ (n)$ = Total time it takes to poll target n
t_{oh} = Time it takes for the poller to prepare an OID query
t_{poll} = Time it takes to complete a poll based on network latency and target resources
t_{db} = Time it takes to write a query response to the db
p_{max} = Timeout period before giving up on processing a poll response

In the above function, t_{oh} will usually be negligible, while the other variables can all be quite significant. Generally, the worst case occurs when an SNMP request times out and no data is inserted into the database. In the more desirable case, the timeout is not reached, but an entire round trip to the device is required in t_{poll}, and a separate round trip to the database is required in t_{db}. In addition, there will be overhead incurred when the SNMP server processes the GET request, as well as when the database server processes the insert. This function must then be applied

[2] RTG2 Issue Tracker: https://code.google.com/p/rtg2/issues/list

to every single target OID being polled by RTG for every poll interval. Given this, it is possible to approximate the cumulative poll time $T_{session}$ to be:

$$T_{session} = \sum_{n \in S} \frac{t(n)}{u}$$

Figure 8 - Approximate cumulative poll time

S = The entire set of OIDs that we must poll
u = The number of threads rtgpoll has been configured to issue polls from

This is only an estimate, as the main thread of RTG will randomize which target is polled by which thread, and one cannot be certain of which particular poll requests will timeout. In the worst case, all threads will be blocking simultaneously on targets that will timeout, preventing productive SNMP request from running. In the case where $T_{session}$ grows to exceed the defined polling interval (which is often set to 300 seconds in real world installations), **rtgpoll** will no longer be able to record OID values as requested in the database. In the most extreme circumstance, this could lead to all iterations of a particular poll exceeding the *HighSkewSlop* and therefore never inserting calculated values into the database.

As an installation of RTG continues to grow, a network operator is at an increased risk of polling being unable to complete within the time constraints of a configured polling interval. There are two primary reasons that $T_{session}$ can increase in a large deployment. Most obviously, the accumulation of targets over time will naturally bring the entire poll time up. Additionally, as the number of targets that are unreachable increases, more polls will timeout, using up valuable portions of the poll interval. This is particularly prevalent in large deployments where it might be commonplace for cards, or even entire routers, to go up and down on a daily basis.

There are several options that one might begin to consider in order to mitigate the effects of an increased poll time as their deployment scales to this level. As a stopgap measure, it may be worth considering ensuring that as few polling requests are timing out as possible. The first line of defense would be to run **rtgtargmkr.pl** on a regular basis to try and account for the reconfiguration of routers. This works

well for small deployments, but may not scale to very large deployments, especially for devices that may be far away from where **rtgtargmkr.pl** is running and latency is high. As an alternative, **rtgtargmkr.pl** could be tailored to an individual's environment or only run on certain volatile devices.

Another option to cut back on poll time is to simply increase the number of threads that **rtgpoll** spawns. While this may scale to a point, it will also increase the number of concurrent database connections. Threads will all be polling and running individual inserts against the database at approximately the same time, which can tax even the most capable of database servers. It is worth reiterating that even with the addition of multiple threads, each database insert requires its own round trip time to the database server.

Finally, as a more extreme measure, a user may decide to deploy an additional instance of RTG. This could allow an operator to strategically assign which RTG instances poll which network devices, and balance resource effectively. Unfortunately, this adds additional overhead for the operator, and introduces a more complex infrastructure into the network environment.

Resiliency of RTG

RTG as a package does not include any tools or options to provide resiliency in the event of failure. In considering the design for a replacement system, it may be desirable to architect it in such a way as to allow resiliency at a base level where convenient. Fortunately, a typical deployment of RTG does not have very many discrete points that may fail, so it is relatively simple to enumerate them.

- **Failure of a network device** – In the event of a network device failing, RTG will clearly be unable to poll values from the SNMP server on that device. This is actually a primary use case for RTG, so no behavior should be modified to account for this scenario.
- **Failure of the poller** – In the current version of RTG, if the RTG poller goes down then no information will be collected from targets configured on that

poller. It is up to the operator of the system to fix the error or manually move the configuration over to a new poller.

- **Failure of a link between the poller and a network device** – If a network device becomes inaccessible to a poller due to a bad network link then it will appear as being completely down to the poller. This is not desirable because statistics that should otherwise be recorded will be lost.
- **Failure of the database (or the link to the database)** – In the event of a database failure the **rtgpoll** program will exit. This is suboptimal, as it requires a running database server to start back up; in the meantime statistics will be completely lost.

In the current implementation of RTG, network operators are left to their own devices in order to mitigate the consequences of all failures listed above. Handling these scenarios are all potential candidates for improvement in a new implementation of RTG.

Distributed RTG

In an ideal scenario, a new version of RTG would abstract away all complexities incurred when scaling the system and provide fault tolerance, ease of configuration, and minimal manual intervention from an operator. In order to conceptualize such a system, first consider the case of a very basic RTG deployment as seen in Figure 9.

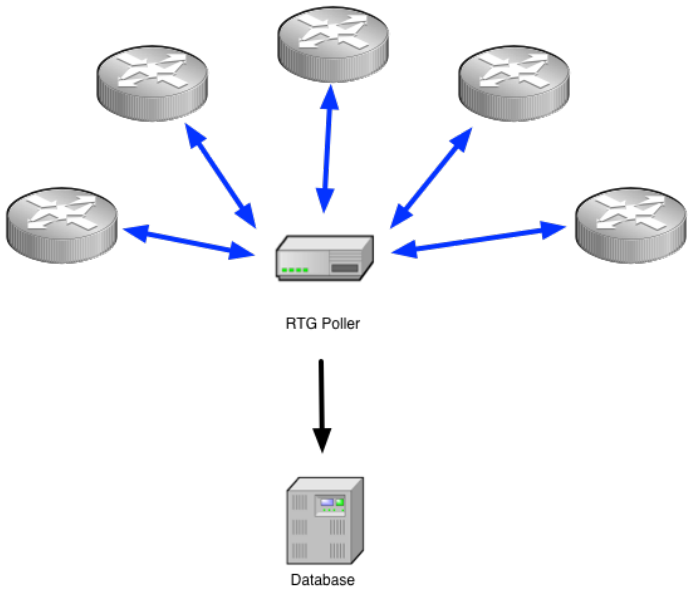

Figure 9 - Basic RTG deployment

Here there is a single RTG poller querying many routers and reporting back to a single database. In the scenario presented above the RTG poller is a single point of failure and, as described previously, may eventually become time constrained as more targets are added. In order to mitigate this, aside from simply increasing the number of threads on the poller, the natural progression is to consider a setup as shown below in Figure 10.

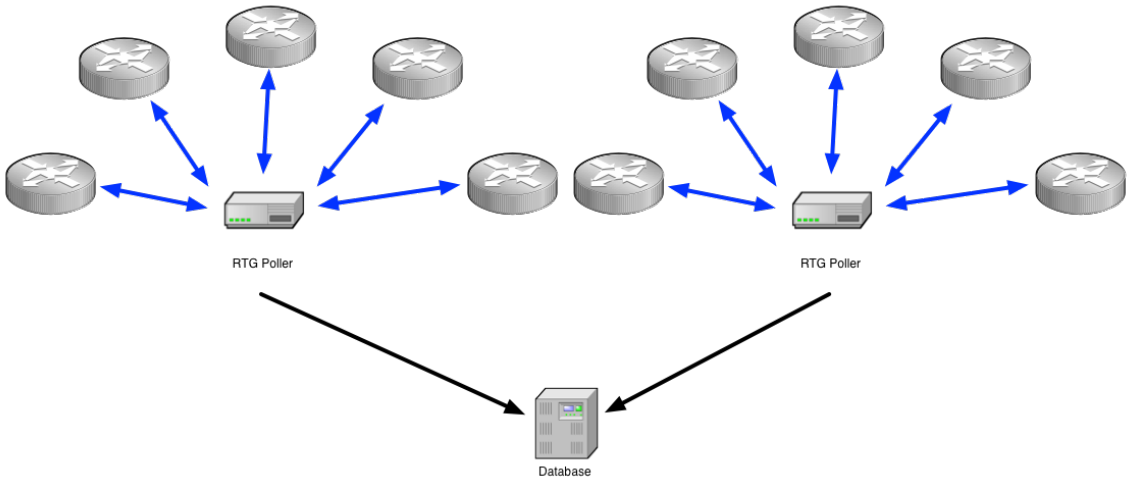

Figure 10 - Adding a second poller

Unfortunately, each RTG poller in this scenario is now its own single point of failure due to the need to individually configure each server. In the event that either poller goes down, it may be possible to migrate polling operations to the other poller, but this would require manual intervention and reconfiguration. Additionally, it is left up to the operator to manually determine which targets are appropriate on which poller every time devices are added or modified.

A better architecture would be one in which each RTG poller could poll any network device, and also know the status of each other RTG poller without manual intervention. This could be accomplished by implementing the architecture depicted in Figure 11.

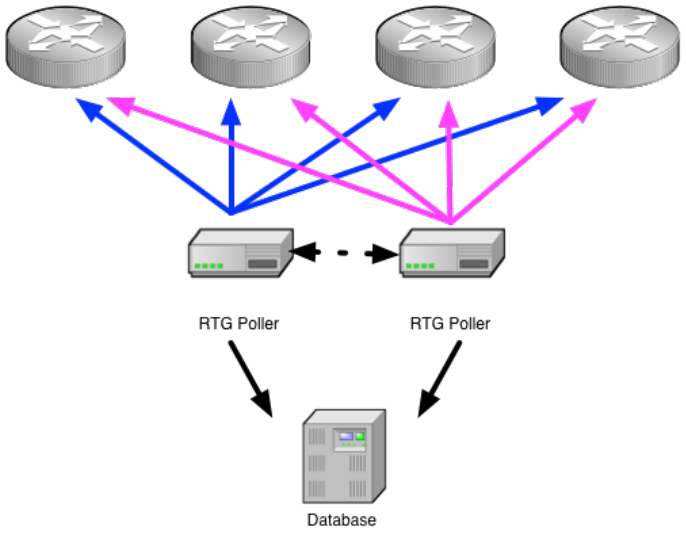

Figure 11 - First distributed RTG proposal

While this solution may appear fairly elegant, in reality it would probably not scale very well. For the scenario in which there are only two RTG pollers, they could easily be configured to know about one another, however if a user would like to add more pollers on top of that, configuration suddenly becomes significantly more difficult. At that point the RTG pollers would need to implement their own peer-to-peer networking algorithm to ensure that they are in sync with each other. While this is not an insurmountable engineering challenge, it adds complexity that may be difficult to fine tune to the particular application at hand. Most notably, this would inherently solve the problem of resiliency, but it would also make the act of maintaining concurrency between each poller a more difficult problem. Suddenly there needs to be a distributed mechanism for determining which poller will be assigned to which device, and all pollers must mutually be aware of this decision. Additionally, each poller must be aware of the liveliness of each other poller to implement an effective system for fault recovery.

In order to simplify the process of synchronization between several pollers, one option is to introduce a message broker that dispatches requests to workers and collects results. This greatly simplifies the problem of concurrency, but re-introduces a single point of failure. To mitigate this in a manageable way, a failover mechanism could be introduced allowing for a reasonably easy to implement concurrency model while also providing an acceptable level of resiliency. This proposed model is illustrated below in Figure 12.

ZeroMQ (ØMQ)

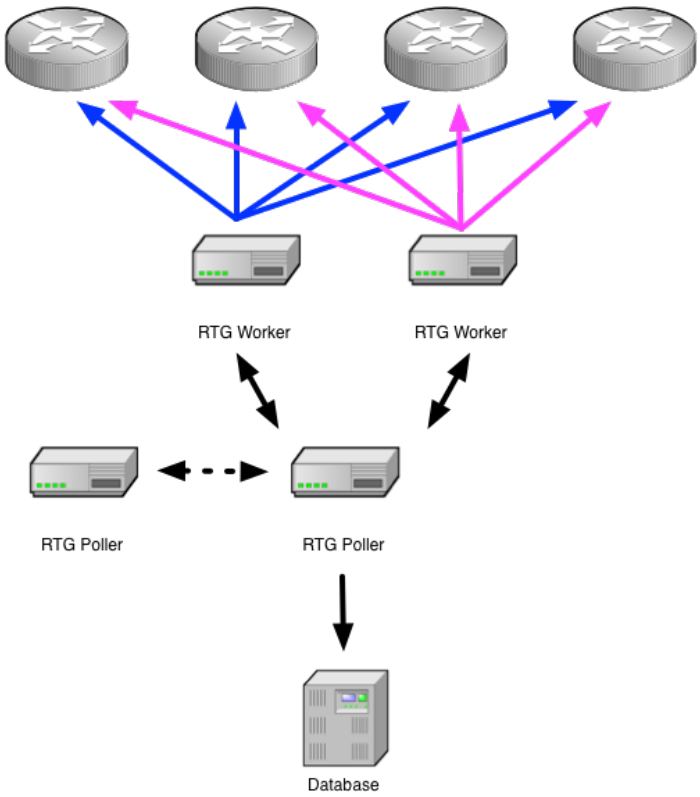

Figure 12 - RTG with failover message broker

Traditionally, implementing a distributed system has been a non-trivial task, requiring a great deal of planning to provide proper concurrency. Fortunately, the past several years have given way to many new design patterns and well-documented implementations of distributed systems. There are now several libraries that successfully abstract away many of the difficulties that accompany the process of writing concurrency into distributed systems.

This project proposes the use of ZeroMQ (ØMQ) for implementing concurrency into a newly designed version of RTG. While the evolution from the current RTG system depicted in Figure 9 to the proposed system in Figure 12 follows a natural, logical progression, so does the evolution from a traditional multithreaded system such as **rtgpoll** to a loosely coupled message passing system as provided by ØMQ. In order to justify this, consider the pieces necessary to architect the solution depicted in Figure 12.

1. The RTG poller should be able to pass messages to other RTG pollers and also to workers.
2. Workers should be able to come and go without any manual reconfiguration.
3. RTG workers should be able to poll multiple routers simultaneous and relay that information back to a poller.

Without the use of a message queuing framework, one would be required to build each of the three bullet points above from scratch. Rather than reinvent the wheel, one can utilize ØMQ sockets in order to perform all of these tasks.

ZeroMQ was created as a product of lessons learned from design of the Advanced Message Queuing Protocol (AMQP). Beginning in 2003 John O'Hara started putting together the first iteration of AMQP while working at JPMorgan Chase in London. It was an open, cooperative effort designed with iMatix and eventually handed off to the AMQP working group in order to get input from other stakeholders (O'Hara, 2007). AMQP has gained significant popularity over the years and is now used to solve a great deal of middleware problems. While AMQP continues to be a very good solution for generic message passing systems, ØMQ takes a different approach and tends to shine in low-level "roll-your-own" solutions to message passing. Whereas AMQP generally implements a standardized message broker in order to orchestrate the movement of messages through a system, ØMQ instead exposes a library of broker-like functionality that can be included into an application to build custom brokers (iMatix Corporation, 2011), making it ideal for a customized application like RTG.

By incorporating ØMQ into a distributed application, developers are able to make use of design patterns that are well documented and supported by the framework. This includes both message passing over TCP, as well as effective implementations of multithreaded paradigms that abstract away the need for mutexes, locks, and other traditional inter-thread communication mechanisms.

ZRTG

This project introduces a proof of concept, drop-in replacement for RTG, dubbed ZRTG. While the program provided should not be run in a production environment, it offers a solid basis toward the development of a truly production worthy implementation. In order to rapidly prototype a working application, ZRTG has been put together using Perl and the ZMQ-LibZMQ3 bindings.

In keeping with the spirit of ØMQ, Figure 13 depicts a model of the ZRTG architecture in a format reminiscent of the examples provided in the ØMQ Guide[3]. Although ZRTG is exposed as a single executable script (**zrtg.pl**), it offers two discrete modes when it is run: a server mode and a worker mode. In a typical deployment, a single instance of ZRTG will be run as a server, while any number of worker instances can be run on different machines. ZRTG then accomplishes a division of work by implementing a ventilator and sink pattern, where a ventilator is in charge of identifying and splitting up work to be sent out to workers and the sink is in charge of aggregating the processed data. Unlike the more typical implementation of a ventilator and sink pattern, ZRTG requires that both of these pieces run within the same process in order to share information.

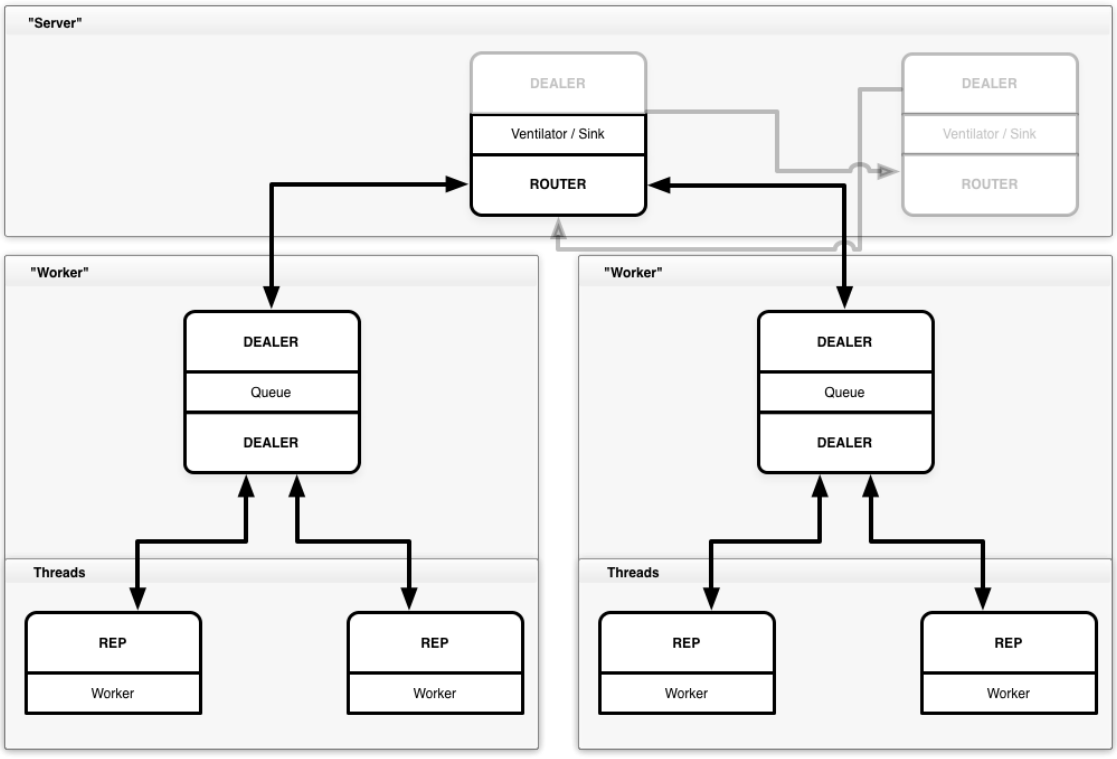

Figure 13 - ZRTG ZeroMQ architecture

[3] ØMQ - The Guide: http://zguide.zeromq.org/

Although ZRTG is implemented in an asynchronous manner, a standard workflow of ZRTG can be roughly simplified to the following:

1. A server starts up and waits for connections from one or more workers.
2. Workers start up and instantiate a user defined number of threads.
3. The ventilator recognizes that the user provided amount of time has elapsed since some targets have been polled. It groups the targets due for polling by network device and dispatches messages to workers, requesting that polling be performed.
4. A worker receives a message requesting some sort of operation (generally polling) to be performed and fair queues it amongst its worker threads.
5. Worker threads perform the assigned task, serialize the results, and pass it back to the parent worker process.
6. The parent worker process sends the response upstream into the sink of the server process.
7. The server checks the sink for data, and if anything needs to be written to a database it is put onto a queue for bulk insertion at a later time.
8. Periodically, the main server process will check the database insertion queue and perform a bulk update of the database.

While the proposed design offers many discrete enhancements, they are almost all byproducts of the gains realized by decoupling the highly serialized execution of **rtgpoll** into ad hoc messages and tasks performed in response to those messages. In order to enhance functionality in the future, it becomes a simple exercise of designing new messages, and injecting logic for handling those messages at any of the points shown in Figure 13. To further demonstrate the current functionality of ZRTG, consider the messages that have already been implemented.

ZRTG_READY – This message is sent from the worker process to a server in order to identify itself to the server. It is also used to affirm that a worker process is still alive and ready to receive polling requests.

ZRTG_PING – This message is sent from the server process to a worker in order to ensure that it is still alive and able to accept polling requests. This is typically only sent if no messages have been received from a worker for a certain period of time, and the absence of a ZRTG_READY message in response to ZRTG_PING will cause the server to remove the target worker from its list of alive workers.

ZRTG_SNMP_SMART_REQUEST – This message is sent by a server to a worker requesting that the worker poll SNMP data from a target device. In the current implementation a ZRTG_SNMP_SMART_REQUEST provides a base OID where walking the subtree would provide counter values for each interface.

ZRTG_SNMP_RESPONSE – This message is sent by a worker to a server as a response to a ZRTG_SNMP_SMART_REQUEST. The response includes a JSON payload that contains responses for all SNMP counter values received in response to a poll. Additionally, this message will contain the latency between this host and the device that it polled in order to be used by the server process during load balancing.

ZRTG_REFRESH_INTERFACES – This message is sent from a server to a worker in order to request all information related to interface names and aliases be queried and returned (this is not yet fully implemented).

ZRTG_INTERFACE_RESPONSE – This message is sent from a worker to a server in response to a ZRTG_REFRESH_INTERFACES request (this is not yet fully implemented).

ZRTG_TAKEOVER_REQUEST – This message is sent from a server to another server in order to initiate a takeover. When a server receives this request it will stop issuing new polling requests, wait for outstanding poll responses, serialize its target hash, issue a ZRTG_TAKEOVER_RESPONSE and shut down gracefully.

ZRTG_TAKEOVER_RESPONSE – This message is sent from a server in response to a ZRTG_TAKEOVER_REQUEST. It will contain a JSON payload containing a serialized version of the targets hash.

As is apparent from the message type descriptions shown above, ZRTG already implements a great deal of functionality to improve the overall scalability and

resiliency of the system in comparison to **rtgpoll**. Some of the key new features include:

- The ability for workers to come and go at any time without configuration on the server side.
- The ability to initiate a manual failover from one server to another without losing any polling data.
- SNMP GETBULK requests for devices.
- Automatic database insertion for new target interfaces.
- A load-balancing algorithm for assigning tasks to workers. In the current implementation a worker is chosen based on the following logic:
    - If no workers have ever polled a target device, choose a random worker.
    - If some workers have not polled a target device, choose one of those workers.
    - If all workers have polled a target device, always choose the worker whose most recent poll to the target device yielded the lowest latency based on round trip time of a single SNMP GET request.

Discussion and Conclusions

While RTG is an enormously popular piece of software that has proven itself over many years, there is still much room for improvement. ZRTG offers an alternative architecture that could greatly improve the resiliency and scalability compared with what RTG currently provides. Although ZRTG is not presented here in a production-ready state, it offers a solid foundation for the future of RTG. In order to prepare ZRTG for production readiness, the following additional review is suggested.

Long-term stability – ZRTG has been tested against 100 routers at a 30 second polling interval for several hours and performed as expected. The long-term ramifications of running under this load are unknown, though. There have traditionally been concerns running multithreaded Perl code in a 24/7 capacity, and

it may be wise to consider porting ZRTG to a compiled language such as C. Fortunately, due to ZRTG being designed around ØMQ, it becomes relatively simple to port into any language in which ØMQ implements a language binding[4].

Security concerns – ZRTG does not consider any measures to ensure the secure connection of workers to servers, or to obfuscate the data flowing between them. All traffic is sent unencrypted via ZMTP (ZeroMQ Message Transport Protocol), often over TCP. One rudimentary option to implement secure connections would be to tunnel traffic over SSH. Alternatively, a future implementation could make use of recent security enhancements made to ØMQ. CurveZMQ is an authentication and encryption protocol for ØMQ that provides fast and secure elliptic-curve cryptography (iMatix Corporation, 2013). Additionally, ØMQ version 4 implements its own security framework with strong encryption based off of CurveZMQ.

Configuration support – As provided, ZRTG does not provide a clean configuration mechanism and instead reads all configuration parameters from the included ZRTG::Config Perl module. While it does provide useful features such as SNMP GETBULK functionality that ultimately allow the configuration of targets to be represented much more concisely (and thus reducing the importance of a program like **rtgtargmkr.pl**), the included solution is still not appropriate for production software.

Database automation – ZRTG attempts to automate some of the database updates that would normally fall in the realm of **rtgtargmkr.pl**. For example, if a new interface is found while bulk walking an SNMP tree, the sink is capable of identifying this and creating new interfaces in the database automatically. This logic needs to be cleaned up, and more tightly integrated with the ZRTG_REFRESH_INTERFACES and ZRTG_INTERFACE_RESPONSE messages outlined in the ZRTG section.

Database failover – The database still represents a single point of failover in the current implementation of ZRTG. Instead, ZRTG should be enhanced to support a

---

[4] ØMQ Language Bindings: http://zeromq.org/bindings:_start

secondary storage mechanism, such as writing to a file, in the case of its primary storage mechanism becoming unavailable.

This project makes significant strides toward the implementation of a system that can truly be used as an RTG replacement. With the proper development resources, and the building blocks provided here, a production version of ZRTG is certainly an attainable goal.

Appendix A - Works Cited

Beverly, R. (2002, 08 7). *[rtg] RTG Version 0.7 Released*. Retrieved 04 04, 2014, from The RTG Archives: http://lists.grdata.com/pipermail/rtg/2002-August/000004.html

Beverly, R. (2007, March 9). *RTG: Real Traffic Grabber*. Retrieved February 12, 2014, from RTG Website: http://rtg.sourceforge.net/

Beverly, R. (2003, 9 24). *rtgpoll*. Retrieved 3 13, 2014, from RTG: http://rtg.sourceforge.net/man/rtgpoll.html

iMatix Corporation. (2013). *CurveZMQ - Security for ZeroMQ*. Retrieved 04 24, 2014, from CurveZMQ: http://curvezmq.org/

iMatix Corporation. (2013, 09 20). *Release notes for ØMQ/4.0.0*. Retrieved 04 24, 2014, from ZeroMQ: http://zeromq.org/docs:changes-4-0-0

iMatix Corporation. (2011, 07 14). *Welcome from AMQP*. Retrieved 04 23, 2014, from ZeroMQ: http://zeromq.org/docs:welcome-from-amqp

iMatix Corporation. (2014). *ØMQ - The Guide*. Retrieved 04 23, 2014, from Zguide: http://zguide.zeromq.org/

Matsudaira, K. (2012). Scalable Web Architecture and Distributed Systems. In A. B. Wilson, *The Architecture of Open Source Applications* (Vol. 2). Lulu.

MySQL. (2001, 10 31). *Changes in Release 3.23.44 (31 October 2001)*. Retrieved 4 6, 2013, from MySQL: http://dev.mysql.com/doc/refman/4.1/en/news-3-23-44.html

O'Hara, J. (2007). Toward a Commodity Enterprise Middleware. *ACM Queue , 5* (4), 48-55.

Appendix B - ZRTG Source Code

ZRTG is implemented as a single Perl script and several packages prefixed with "ZRTG::" as shown in Figure 14. The source code for each individual file is included in the remainder of Appendix B - ZRTG Source Code.

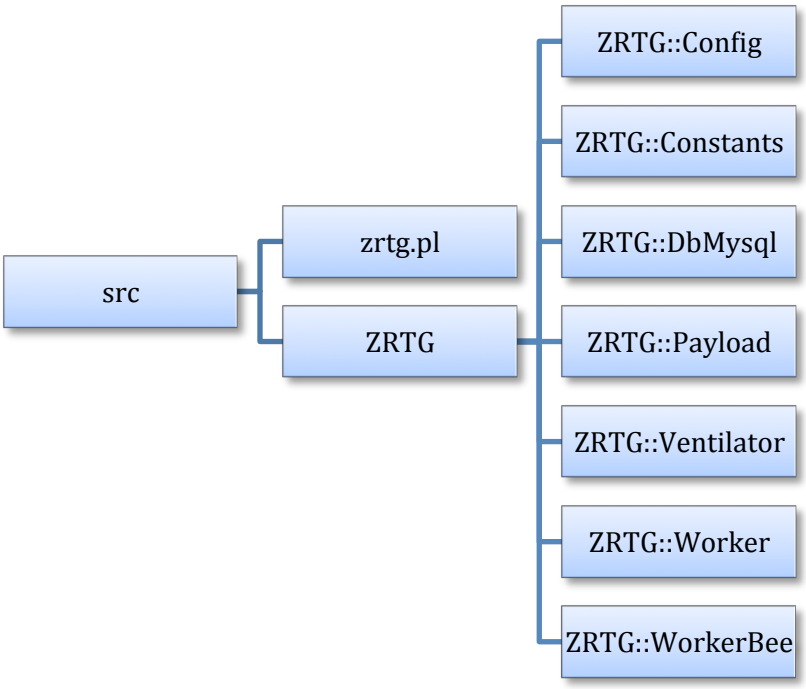

Figure 14 - ZRTG source tree

zrtg.pl

```
#!/usr/bin/perl

use strict;
use warnings;
use 5.10.0;

use ZRTG::Config;

use Getopt::Long;

my ($debug, $takeover, $worker);

GetOptions ("d|debug"           => \$debug,
            "w|worker=s"        => \$worker,
            "t|takeover=s"      => \$takeover);

if ($debug) {
    $ZRTG::Config::DEBUG = $debug;
}

my $endpoint = shift;

if (! defined($endpoint) ||
      $endpoint !~ /^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}:\d+$/) {
    print "usage: $0 <server:port>\n";
    exit -1;
}

my $obj;

if ($worker) {
    require ZRTG::Worker;
    $obj = ZRTG::Worker->new($worker);
} else {
    require ZRTG::Ventilator;
    $obj = ZRTG::Ventilator->new();
}

if ($takeover) {
    $obj->start("tcp://${endpoint}", $takeover);
} else {
    $obj->start("tcp://${endpoint}");
}
```

ZRTG::Config

```
package ZRTG::Config;

use 5.10.0;

use strict;
use warnings;

our $DB_USER = 'dbuser';
our $DB_PASS = 'dbpass';
our $DB_SCHEMA = 'zrtg';
our $DB_HOST = '127.0.0.1';

our $DEBUG = 0;

our $COMMUNITY = 'public';
our $SNMPVER = 'snmpv2c';

our @HOSTS = qw(10.100.0.1
                10.100.0.2
                10.100.0.3
                10.100.0.4
               );

1;
```

ZRTG::Constants

```
package ZRTG::Constants;
use strict;
use base qw(Exporter);
use Carp ();

my %constants;
BEGIN {
    %constants  = (
        # Message types
        ZRTG_PING                   => 30,
        ZRTG_READY                  => 31,
        ZRTG_SNMP_SMART_REQUEST     => 40,
        ZRTG_SNMP_RESPONSE          => 51,
        ZRTG_REFRESH_INTERFACES     => 60,
        ZRTG_INTERFACE_RESPONSE     => 61,
        ZRTG_TAKEOVER_REQUEST       => 70,
        ZRTG_TAKEOVER_RESPONSE      => 71,

        # SNMP OIDs
        NUM_INTERFACES_OID          => '.1.3.6.1.2.1.2.1.0',
        INTERFACE_HIGHSPEED_OID     => '.1.3.6.1.2.1.31.1.1.1.15',
        INTERFACE_NAMES_OID         => '.1.3.6.1.2.1.31.1.1.1.1',
        INTERFACE_ALIASES_OID       => '.1.3.6.1.2.1.31.1.1.1.18',
        INTERFACE_ADMIN_STATUS_OID  => '.1.3.6.1.2.1.2.2.1.7',
        INTERFACE_OPER_STATUS_OID   => '.1.3.6.1.2.1.2.2.1.8',
    );
}

use constant \%constants;
our @EXPORT;
our @EXPORT_OK = keys %constants;
our %EXPORT_TAGS = (
    all => [ qw(
        ZRTG_PING
        ZRTG_READY
        ZRTG_SNMP_SMART_REQUEST
        ZRTG_SNMP_RESPONSE
        ZRTG_REFRESH_INTERFACES
        ZRTG_INTERFACE_RESPONSE
        ZRTG_TAKEOVER_REQUEST
        ZRTG_TAKEOVER_RESPONSE

        NUM_INTERFACES_OID
        INTERFACE_HIGHSPEED_OID
        INTERFACE_NAMES_OID
        INTERFACE_ALIASES_OID
        INTERFACE_ADMIN_STATUS_OID
        INTERFACE_OPER_STATUS_OID
    ) ]
);

$EXPORT_TAGS{all} = [ @EXPORT_OK ];

1;
```

ZRTG::DbMysql

```
package ZRTG::DbMysql;

use 5.10.0;

use strict;
use warnings;

use DBI;

use ZRTG::Config;

sub new {
    my $class = shift;

    my $self = {
        dbh             => undef,
    };

    bless $self, $class;
}

sub getIid {
    my $self = shift;
    my $host = shift;
    my $ifName = shift;

    if (! $self->{targets}->{$host}->{interfaces}->{$ifName}) {
        # We don't know about this interface yet, look it up

        my $ret = $self->dbh->selectall_arrayref(q/select `id` from `interface` where
rid=? and name=?/, { }, $self->{targets}->{$host}->{rid}, $ifName );

        if (@{$ret}) {
            $self->{targets}->{$host}->{interfaces}->{$ifName} = $ret->[0]->[0];
        } else {
            $self->dbh->do(q/insert into `interface` (`name`, `rid`) values (?, ?)/, {},
$ifName, $self->{targets}->{$host}->{rid});
            $self->{targets}->{$host}->{interfaces}->{$ifName} = $self->dbh-
>{mysql_insertid};
        }
    }

    return $self->{targets}->{$host}->{interfaces}->{$ifName};
}

sub dbh {
    my $self = shift;

    if (! $self->{dbh} ) {
        my $host = $ZRTG::Config::DB_HOST;
        my $user = $ZRTG::Config::DB_USER;
        my $pw = $ZRTG::Config::DB_PASS;
        my $db = $ZRTG::Config::DB_SCHEMA;

        $self->{dbh} = DBI->connect("dbi:mysql:database=$db;host=$host",
                $user,$pw) or die $DBI::errstr;
    }
```

```
    return $self->{dbh};
}

sub insertCounterValues {
    my $self = shift;
    my $table = shift;
    my $chunk = shift;

    # TODO: Fix this so we aren't at risk of SQL injection
    my $query = "insert into `${table}` (`id`, `dtime`, `counter`, `rate`) values ";

    for my $i (0 .. @{$chunk} - 1) {
        $query .= "($chunk->[$i]->[0],'$chunk->[$i]->[1]',$chunk->[$i]->[2],$chunk->[$i]-
>[3])";

        if ($i < @{$chunk} - 1) {
            $query .= ',';
        }
    }

    $self->dbh->do($query);
}

1;
```

ZRTG::Payload

```
package ZRTG::Payload;

use 5.10.0;

use strict;
use warnings;

use Carp;

use ZRTG::Config;
use ZRTG::Constants qw(:all);
use Log::Message::Simple qw(:STD);

use ZMQ::LibZMQ3;
use ZMQ::Constants qw(:all);

sub new {
    my $class = shift;
    my $arg = shift;
    my @data = @_;

    my $self = { data => (@_) ? [@_] : [] };

    if (ref($arg)) {
        # Read in all of our data (block)
        my $msg;

        while (1) {
            my $msg = zmq_msg_init();
            my $data;

            zmq_msg_recv($msg, $arg);
            $data = zmq_msg_data($msg);

            if (! $self->{type}) {
                if ($data) {
                    $self->{type} = $data;
                } else {
                    print "No type yet, and our data frame is empty!\n";
                }
            } else {
                push @{$self->{data}}, $data;
            }

            if (! zmq_getsockopt($arg, ZMQ_RCVMORE)) {
                last;
            }
        }
    } else {
        $self->{type} = $arg;
    }

    bless $self, $class;
}

sub setData {
    my $self = shift;
```

```
    my @data = @_;

    push @{$self->{data}}, @data;
}

sub send {
    my $self = shift;
    my $socket = shift;
    my $destination = shift;

    if (! ref($socket) || ! $socket->isa('ZMQ::LibZMQ3::Socket')) {
        carp("Expected a ZMQ::LibZMQ3::Socket object, bailing");
        return;
    }

    if ($self->{type} == ZRTG_READY) {
        debug "Sending READY", $ZRTG::Config::DEBUG;

    } elsif ($self->{type} == ZRTG_PING) {
        debug "Sending PING to $destination", $ZRTG::Config::DEBUG;

    } elsif ($self->{type} == ZRTG_SNMP_SMART_REQUEST) {
        debug "Sending request to $destination: $self->{data}->[0]",
$ZRTG::Config::DEBUG;

    } elsif ($self->{type} == ZRTG_SNMP_RESPONSE) {
        debug "Sending SNMP response", $ZRTG::Config::DEBUG;

    } elsif ($self->{type} == ZRTG_REFRESH_INTERFACES) {
        debug "Requesting interface refresh", $ZRTG::Config::DEBUG;

    } elsif ($self->{type} == ZRTG_INTERFACE_RESPONSE) {
        debug "Sending interface refresh response", $ZRTG::Config::DEBUG;

    } elsif ($self->{type} == ZRTG_TAKEOVER_REQUEST) {
        debug "Sending takeover request", $ZRTG::Config::DEBUG;

    } elsif ($self->{type} == ZRTG_TAKEOVER_RESPONSE) {
        debug "Sending takeover response", $ZRTG::Config::DEBUG;

        zmq_send($socket, $destination, -1, ZMQ_SNDMORE);

    } else {
        say "Unknown message type: $self->{type}";
        return undef;
    }

    $self->_sendAll($socket, $destination);
}

sub _sendAll {
    my $self = shift;
    my $socket = shift;
    my $destination = shift;

    if (! defined($self->{type})) {
        carp "No type defined for payload";
        return -1;
    }
```

```
    if (defined($destination)) {
        zmq_send($socket, $destination, -1, ZMQ_SNDMORE);
    }

    if (@{$self->{data}} == 0) {
        zmq_send($socket, $self->{type}, -1);
    } elsif (@{$self->{data}} == 1) {
        zmq_send($socket, $self->{type}, -1, ZMQ_SNDMORE);
        zmq_send($socket, $self->{data}->[0], -1)
    } else {
        zmq_send($socket, $self->{type}, -1, ZMQ_SNDMORE);

        for (my $i = 0; $i < @{$self->{data}} - 1; $i++) {
            zmq_send($socket, $self->{data}->[$i], -1, ZMQ_SNDMORE);
        }
        zmq_send($socket, $self->{data}->[-1]);
    }
}

1;
```

ZRTG::Ventilator

```
package ZRTG::Ventilator;

use 5.10.0;

use strict;
use warnings;

# Various necessary libraries
use Carp;
use JSON;
use Time::HiRes qw(gettimeofday tv_interval);
use Log::Message::Simple qw(:STD);
use POSIX qw(strftime);

# ZMQ Libraries
use ZMQ::LibZMQ3;
use ZMQ::Constants qw(:all);

# ZRTG Libraries
use ZRTG::Constants qw(:all);
use ZRTG::Config;
use ZRTG::Payload;
use ZRTG::DbMysql;

sub new {
    my $class = shift;

    my $self = {
        db                  => ZRTG::DbMysql->new(),
        insertBuffer        => {},
        context             => zmq_ctx_new(),
        debug               => $ZRTG::Config::DEBUG,
        workerSocket        => undef,
        targets             => {},
        workers             => {},
        startupTime         => 1.5,
        pollInterval        => 30,                          # Seconds
        blockInterval       => 100,                         # Milliseconds
        timeFormat          => '%Y-%m-%d %H:%M:%S',
        insertBatchSize     => 50,
        pruneThreshold      => 20,
        shutdownWait        => 2000,                        # Milliseconds
        readyToPoll         => 1,
    };

    bless $self, $class;

    $self->{db}->{targets} = $self->{targets};

    return $self;
}

sub DESTROY {
    my $self = shift;

    if ( $self->{workerSocket} ) {
        zmq_close( $self->{workerSocket} );
```

```
    }

    if ( $self->{context} ) {
        zmq_ctx_destroy( $self->{context} );
    }
}

sub connectWorker {
    my $self = shift;
    my $address = shift;

    my $workerSocket = zmq_socket( $self->{context}, ZMQ_ROUTER );

    if (! defined($workerSocket)) {
        die "Could not create worker socket: $!";
    }

    if (0 != zmq_bind( $workerSocket, $address )) {
        die "Could not bind worker socket ($address): $!";
    }

    $self->{workerSocket} = $workerSocket;
}

sub start {
    my $self = shift;
    my $workerPort = shift;
    my $takeoverFrom = shift;

    $self->loadConfig();

    debug "Connecting to worker socket", $self->{debug};

    $self->connectWorker($workerPort);

    my $dealerItem = {
        socket   => $self->{workerSocket},
        events   => ZMQ_POLLIN,
        callback => sub {
            sink( $self );
        },
    };


    # Don't need to wait if we are going to go asking to takeover
    # from another ventilator, but we should make sure somewhere that
    # we ignore children until then

    if (! $takeoverFrom) {
        # Give us some time to listen for workers before we
        # start shelling out orders.
        my $t0 = [gettimeofday];

        while ( tv_interval($t0) < $self->{startupTime} ) {
            zmq_poll( [ $dealerItem ], 0)
        }
    }

    # Alright, let's go
    while (1) {
```

```
        if (defined($takeoverFrom)) {
            # Initiate the takeover request
            $self->requestTakeover($takeoverFrom, $workerPort);

            # Don't poll anything until we've heard back
            $self->{readyToPoll} = 0;

            # Don't try and make another takeover request
            $takeoverFrom = undef;
        }

        my $events = zmq_poll( [ $dealerItem ], $self->{blockInterval} );

        $self->pruneWorkers();

        if ($self->{readyToPoll}) {
            $self->makeRequests();
        }

        $self->processInserts();


    }
}

sub requestTakeover {
    my $self = shift;
    my $from = shift;
    my $workerPort = shift;

    my $requestSocket = zmq_socket( $self->{context}, ZMQ_DEALER );

    if (0 != zmq_connect( $requestSocket, "tcp://${from}" )) {
        die "Could not connect to peer ventilator ($from): $!";
    }

    debug "Requesting to takeover from ventilator at $from", $self->{debug};

    zmq_send($requestSocket, $from, 0, ZMQ_SNDMORE);
    zmq_send($requestSocket, ZRTG_TAKEOVER_REQUEST, -1, ZMQ_SNDMORE);
    zmq_send($requestSocket, $workerPort, -1);
}

sub pruneWorkers {
    my $self = shift;

    for my $workerId (keys(%{$self->{workers}})) {
        if ($self->{workers}->{$workerId}->{alive} == 0) {
            # We've seen you before, time for the boot
            debug "Worker $workerId is getting the boot", $self->{debug};
            delete($self->{workers}->{$workerId});
        } elsif ((time() - $self->{workers}->{$workerId}->{alive}) > $self-
>{pruneThreshold}) {
            my $payload = ZRTG::Payload->new(ZRTG_PING);

            # Put this guy in the dog house
            $self->{workers}->{$workerId}->{alive} = 0;

            $payload->send($self->{workerSocket}, $workerId);
        }
```

```
    }
}

# Pass on flags if they are sent in
sub getMessage {
    my $self = shift;

    my $msg = zmq_recvmsg( $self->{workerSocket}, @_ );

    if (! defined($msg)) {
        error "Error receiving message: $!"
    }

    return $msg;
}

# Pass on flags if they are sent in
sub getData {
    my $self = shift;

    my $msg = $self->getMessage( @_ );

    if (defined($msg)) {
        return zmq_msg_data( $msg );
    }

    return undef;
}

sub sink {
    my $self = shift;

    # Things coming into the sink will be prefaced with a workerId and empty
    # frame
    my $workerId = $self->getData();
    my $empty = $self->getData();

    # If there was no empty frame... something is wrong
    if ($empty) {
        carp "Expected empty frame from but got $empty";
        return;
    }

    # It should be safe to receive the payload after that
    my $payload = ZRTG::Payload->new( $self->{workerSocket} );

    if ($payload->{type} ne ZRTG_TAKEOVER_REQUEST &&
        $payload->{type} ne ZRTG_TAKEOVER_RESPONSE) {
        # Record that the worker isn't dead
        $self->{workers}->{$workerId}->{alive} = time();
    }

    if ($payload->{type} eq ZRTG_READY) {
        debug "Worker $workerId is ready", $self->{debug};
    } elsif ($payload->{type} eq ZRTG_SNMP_RESPONSE) {
        $self->processIncomingCounters( $payload, $workerId );

    } elsif ($payload->{type} eq ZRTG_INTERFACE_RESPONSE) {
        $self->processIncomingInterfaces( $payload, $workerId );
```

```
    } elsif ($payload->{type} eq ZRTG_TAKEOVER_REQUEST) {
        $self->processTakeoverRequest( $payload, $workerId );

    } elsif ($payload->{type} eq ZRTG_TAKEOVER_RESPONSE) {
        $self->processTakeoverResponse( $payload, $workerId );

    } else {
        say "Unknown type $payload->{type}";
    }
}

sub processTakeoverResponse {
    my $self = shift;
    my $payload = shift;
    my $from = shift;

    my $newTargets = decode_json($payload->{data}->[0]);

    $self->{targets} = $newTargets;

    # We're good to go now!
    $self->{readyToPoll} = 1;
}

sub processTakeoverRequest {
    my $self = shift;
    my $payload = shift;
    my $newGuy = shift;

    my $connectTo = $payload->{data}->[0];

    my $replySocket = zmq_socket( $self->{context}, ZMQ_DEALER );

    debug "Takeover request received.  Handing over to ventilator at $connectTo", $self-
>{debug};

    # Connect to the peer ventilator who is taking over
    if (0 != zmq_connect( $replySocket, $connectTo )) {
        warn "Could not connect to peer ventilator ($connectTo): $!";
    }

    # Send our targets hash to them
    zmq_send($replySocket, '', 0, ZMQ_SNDMORE);
    zmq_send($replySocket, ZRTG_TAKEOVER_RESPONSE, -1, ZMQ_SNDMORE);
    zmq_send($replySocket, encode_json($self->{targets}), -1);


    # Yuck, copy and pasted from start(), let's make this nicer
    my $dealerItem = {
        socket   => $self->{workerSocket},
        events   => ZMQ_POLLIN,
        callback => sub {
            sink( $self );
        },
    };

    my $events;

    debug "Handed off everything I know, waiting a few for straggling workers", $self-
>{debug};
```

```
    do {
        # Wait a little bit to see if workers are still shelling out responses
        # to us
        $events = zmq_poll( [ $dealerItem ], $self->{shutdownWait} );

        $self->processInserts();
    } while ($events);

    debug "Looks good, shutting down", $self->{debug};

    # Clean up after ourselves
    zmq_close( $replySocket );

    # Game over folks
    exit 0;
}

sub chooseWorker {
    my $self = shift;
    my $host = shift;
    my $method = shift;

    my @workers = grep({ $self->{workers}->{$_}->{alive} ne 0 } keys(%{$self-
>{workers}}));

    my $winner;

    # These choices are easy
    if (@workers == 0) {
        return undef;
    } elsif (@workers == 1) {
        return $workers[0];
    }

    # For these we need to do something at least a little intelligent
    if ($method eq 'latency') {
        # [ worker, latency ]
        my @cheapestWorker;

        # If we find any workers who have never polled, we want to try them
        # ASAP to see if they have a better connection
        my $slacker;

        for my $worker (@workers) {
            if ($self->{workers}->{$worker}->{latency}->{$host}) {
                my $latency = $self->{workers}->{$worker}->{latency}->{$host};

                if (! $cheapestWorker[1]) {
                    @cheapestWorker = ( $worker, $latency );
                } elsif ($cheapestWorker[1] > $latency) {
                    @cheapestWorker = ( $worker, $latency );
                }
            } else {
                $slacker = $worker;
            }
        }

        # I guess the slacker always wins?
        if ($slacker) {
```

```
            $winner = $slacker;
        } elsif (@cheapestWorker) {
            $winner = $cheapestWorker[0];
        }
    }

    if (! $winner) {
        # The good old fallback: choose a random worker
        $winner = $workers[rand() * 10 % scalar(@workers)]
    }

    return $winner;
}

#
# Returns a hash:
# $poller -> $host -> targetgroups -> $oid
sub needsPolling {
    my $self = shift;

    my $targets = $self->{targets};
    my %ret;
    my @pollers = keys(%{$self->{workers}});

    if (@pollers) {
        for my $host (keys(%{$targets})) {
            # Use latency based load balancing
            my $poller;

            for my $targetgroup (keys(%{$targets->{$host}->{targetgroups}})) {
                if (defined($targets->{$host}->{targetgroups}->{$targetgroup}-
>{lastPolled})) {
                    my $time = time();
                    my $last = $targets->{$host}->{targetgroups}->{$targetgroup}-
>{lastPolled};

                    if ($time - $last > $self->{pollInterval}) {
                        debug "$host:$targetgroup was last polled at $last, but it's
$time", $self->{debug};
                        if (! $poller) {
                            # Only do this calculation once we know we are going to poll
                            $poller = $self->chooseWorker($host, 'latency');
                        }
                        push @{$ret{$poller}->{$host}->{targetgroups}}, $targetgroup;
                    }
                } else {
                    if (! $poller) {
                        # Only do this calculation once we know we are going to poll
                        $poller = $self->chooseWorker($host, 'latency');
                    }
                    push @{$ret{$poller}->{$host}->{targetgroups}}, $targetgroup;
                }
            }
        }
    }

    return \%ret;
}

sub createRefreshRequest {
```

```
    my $self = shift;
    my $rid = shift;

    my $payload;

    for my $host (keys(%{$self->{targets}})) {
        if ($self->{targets}->{$host}->{rid} eq $rid) {
            $payload = ZRTG::Payload->new(ZRTG_REFRESH_INTERFACES);
            $payload->setData($host,
                              $self->{targets}->{$host}->{community},
                              $self->{targets}->{$host}->{snmpver}
                             );
            return $payload;
        }
    }

    return undef;
}

sub processInserts {
    my $self = shift;

    for my $table (keys(%{$self->{insertBuffer}})) {
        while (my @chunk = splice(@{$self->{insertBuffer}->{$table}}, 0, $self-
>{insertBatchSize})) {
            $self->{db}->insertCounterValues($table, \@chunk);
        }
    }
}

sub makeRequests {
    my $self = shift;

    my $needsPolling = $self->needsPolling();

    for my $poller (keys(%{$needsPolling})) {
        for my $targetHost (keys(%{$needsPolling->{$poller}})) {
            my @targetGroups = @{$needsPolling->{$poller}->{$targetHost}-
>{targetgroups}};

            if (@targetGroups) {
                # We have entire groups that need polling, make SNMP "smart requests""
                my $payload = ZRTG::Payload->new(ZRTG_SNMP_SMART_REQUEST);

                $payload->setData(
                    $targetHost,
                    $self->{targets}->{$targetHost}->{community},
                    $self->{targets}->{$targetHost}->{snmpver},
                    @targetGroups,
                );

                $payload->send($self->{workerSocket}, $poller);

                my $time = time();

                for my $targetGroup (@{$needsPolling->{$poller}->{$targetHost}-
>{targetgroups}}) {
                    $self->{targets}->{$targetHost}->{targetgroups}->{$targetGroup}-
>{lastPolled} = $time;
                }
```

```
            }

            # We could make some less smart requests here, if we cared to implement that
        }
    }
}

sub loadConfig {
    my $self = shift;

    my @hosts = @ZRTG::Config::HOSTS;

    for my $i (1 .. @hosts) {
        my $host = $hosts[$i - 1];

        $self->{targets}->{$host}->{community} = $ZRTG::Config::COMMUNITY;
        $self->{targets}->{$host}->{snmpver} = $ZRTG::Config::SNMPVER;
        $self->{targets}->{$host}->{rid} = $i;                  #### THIS IS NEW ###
        $self->{targets}->{$host}->{targetgroups}->{'.1.3.6.1.2.1.31.1.1.1.7'} = { #
ifHCInUcastPkts
            table   => "ifInUcastPkts_$i",
        };
        $self->{targets}->{$host}->{targetgroups}->{'.1.3.6.1.2.1.31.1.1.1.6'} = { #
ifHCInOctets
            table   => "ifInOctets_$i",
        };
        $self->{targets}->{$host}->{targetgroups}->{'.1.3.6.1.2.1.31.1.1.1.10'} = { #
ifHCOutOctets
            table   => "ifOutOctets_$i",
        };
        $self->{targets}->{$host}->{targetgroups}->{'.1.3.6.1.2.1.31.1.1.1.11'} =  { #
ifHCOutUcastPkts
            table   => "ifOutUcastPkts_$i",
        };
    }
}

sub processIncomingInterfaces {
    my $self = shift;
    my $payload = shift;
    my $workerId = shift;

    my $host = $payload->{data}->[0];
    my $json = $payload->{data}->[1];

    debug "Received interface data from $workerId for $host", $self->{debug};
}

sub processIncomingCounters {
    my $self = shift;
    my $payload = shift;
    my $workerId = shift;

    my $host = $payload->{data}->[0];
    my $latency = unpack('f<', $payload->{data}->[1]);
    my $json = $payload->{data}->[2];

    # Nice even millisecond number
    my $latencyFormatted = int($latency * 1000);
```

```
    # Used for recording when this data came in
    my $time = time();

    debug "Received counter data from $workerId: Latency to host $host was
$latencyFormatted milliseconds, JSON payload (" . length($json) . " bytes) was attached",
$self->{debug};

    $self->{workers}->{$workerId}->{latency}->{$host} = $latency;

    # [ oid, counter, time, type ]
    my $data = decode_json($json);

    my %receivedBaseOids;

    for my $set (@{$data}) {
        # set = [ oid, ifName, counter, dtime, counterType ]

        if ($set->[0] =~ /(.*)(\.\d+)/) {
            my ($base, $interface) = ($1, $2);

            my $iid = $self->{db}->getIid($host, $set->[1]);

            my $table;

            if (exists($self->{targets}->{$host}->{targetgroups}->{$base}->{table})) {
                $table = $self->{targets}->{$host}->{targetgroups}->{$base}->{table};
            } else {
                # This shouldn't really ever happen, it means we got bogus data
                # from the worker
                warn "Could not find table for base OID $base";
                next;
            }

            if (! $iid) {
                # We couldn't look up (or create) an interface, just drop
                # this until we determine something better to do with it
                next;
            }

            $receivedBaseOids{$base}++;

            # counters hash => [ counter, dtime ]

            if (exists($self->{targets}->{$host}->{counters}->{$table}->{$iid})) {
                # We have a previous counter value in format [ counter, time ]
                # Calculate the delta
                my ($lastCounter, $lastTime) = @{$self->{targets}->{$host}->{counters}-
>{$table}->{$iid}};

                my $timeDiff = $set->[3] - $lastTime;
                my $counterDiff = $set->[2] - $lastCounter;

                my $rate = $counterDiff / $timeDiff;

                # Queue it for inserting into the database
                push @{$self->{insertBuffer}->{$table}}, [ $iid, strftime($self-
>{timeFormat}, gmtime($set->[3])), $counterDiff, $rate ]

            } else {
                # First time we've seen this interface, insert 0 value
```

```
                push @{$self->{insertBuffer}->{$table}}, [ $iid, strftime($self-
>{timeFormat}, gmtime($set->[3])), 0, 0 ]
            }

            # Set the counter values in our internal data structure [ counter, time ]
            $self->{targets}->{$host}->{counters}->{$table}->{$iid} = [ $set->[2], $set-
>[3] ];
        } else {
            error "$set->[0] doesn't look like an OID"
        }
    }

    for my $receivedBaseOid (keys(%receivedBaseOids)) {
        $self->{targets}->{$host}->{targetgroups}->{$receivedBaseOid}->{lastReceived} =
$time;
    }
}

1;
```

ZRTG::Worker

```
package ZRTG::Worker;

use 5.10.0;

use threads;

use strict;
use warnings;

# Various necessary libraries
use Carp;
use Log::Message::Simple qw(:STD);

# ZMQ Libraries
use ZMQ::LibZMQ3;
use ZMQ::Constants qw(:all);

# ZRTG Libraries
use ZRTG::Constants qw(:all);
use ZRTG::Config;
use ZRTG::Payload;
use ZRTG::WorkerBee;

sub new {
    my $class = shift;
    my $name = shift;

    my $self = {
        context             => zmq_ctx_new(),
        upstreamSocket      => undef,
        downstreamSocket    => undef,
        debug               => $ZRTG::Config::DEBUG,
        name                => $name,
        numThreads          => 2,
        keepAlive           => 300,                         # Seconds
        blockInterval       => 100,                         # Milliseconds
        lastUpstream        => 0,
    };

    bless $self, $class;
}

sub DESTROY {
    my $self = shift;

    if ( $self->{upstreamSocket} ) {
        zmq_close( $self->{upstreamSocket} );
    }

    if ( $self->{downstreamSocket} ) {
        zmq_close( $self->{downstreamSocket} );
    }

    if ( $self->{context} ) {
        zmq_ctx_destroy( $self->{context} );
    }
}
```

```
sub connectSockets {
    my $self = shift;
    my $upstreamAddress = shift;

    # Start with the upstream socket
    my $upstreamSocket = zmq_socket( $self->{context}, ZMQ_DEALER );

    if (! defined($upstreamSocket)) {
        die "Could not create upstream socket: $!";
    }

    # Set the identity of this worker
    zmq_setsockopt( $upstreamSocket, ZMQ_IDENTITY, $self->{name} );

    if (0 != zmq_connect( $upstreamSocket, $upstreamAddress )) {
        die "Could not connect upstream socket: $!";
    }

    # Looks good, save it
    $self->{upstreamSocket} = $upstreamSocket;

    # Time to do the downstream
    my $downstreamSocket = zmq_socket( $self->{context}, ZMQ_DEALER );

    if (! defined($downstreamSocket)) {
        die "Could not create downstream socket: $!";
    }

    if (0 != zmq_bind( $downstreamSocket, 'inproc://workers' ) ) {
        die "Could not bind downstream socket: $!";
    }

    # Looks good, save it
    $self->{downstreamSocket} = $downstreamSocket;

    return 0;
}

sub start {
    my $self = shift;
    my $port = shift;

    debug "Connecting sockets", $self->{debug};
    $self->connectSockets($port);

    # Launch pool of worker threads
    debug "Creating $self->{numThreads} threads", $self->{debug};
    for ( 1 .. $self->{numThreads} ) {
        threads->create( 'ZRTG::WorkerBee::main', $self->{context}, $self->{debug} );
    }

    # Tell the ventilator that we're ready to roll
    $self->sendReady();

    # Items to poll on
    my $upstreamItem = {
        socket => $self->{upstreamSocket},
        events => ZMQ_POLLIN,
        callback => sub {
```

```
            debug "Received request, forwarding downstream", $self->{debug};
            $self->dealerForwarder('downstream');
        },
    };

    my $downstreamItem = {
        socket => $self->{downstreamSocket},
        events => ZMQ_POLLIN,
        callback => sub {
            debug "Received response, forwarding upstream", $self->{debug};
            $self->dealerForwarder('upstream');
        },
    };

    # Just keep swimming
    while (1) {
        my $events = zmq_poll([ $upstreamItem, $downstreamItem ], $self-
>{blockInterval});

        if ($self->{keepAlive}) {
            if (time() - $self->{lastUpstream} > $self->{keepAlive}) {
                $self->sendReady();
            }
        }
    }

    # We should flush log buffer (to somewhere) periodically
}

# Forward packages up/downstream to workers.  Fair queued by the ZMQ dealer
sub dealerForwarder {
    my $self = shift;
    my $mode = shift;

    my $fromSocket;
    my $toSocket;

    # If mode is upstream then we strip off the leading empty frame
    # If mode is downstream then we add a leading empty frame
    if ($mode eq 'upstream') {
        $fromSocket = $self->{downstreamSocket};
        $toSocket = $self->{upstreamSocket};

        my $empty;
        my $emptyReceived = zmq_recv($fromSocket, $empty, 1);

        if ($emptyReceived) {
            carp "Expected empty frame from downstream but got $empty";
            return;
        } else {
            print "Stripping empty frame\n" if ($self->{debug} > 2);
        }

        $self->{lastUpstream} = time();
    } elsif ($mode eq 'downstream') {
        $fromSocket = $self->{upstreamSocket};
        $toSocket = $self->{downstreamSocket};

        my $typeMessage = zmq_recvmsg($fromSocket);
        my $type = zmq_msg_data($typeMessage);
```

```
        if ($type == ZRTG_PING) {
            # Deal with this immediately
            $self->sendReady();

            return;
        } else {
            print "sending empty frame and more\n"  if ($self->{debug} > 2);
            zmq_send($toSocket, '', 0, ZMQ_SNDMORE);
            print "sending $type and more\n"  if ($self->{debug} > 2);
            zmq_send($toSocket, $type, -1, ZMQ_SNDMORE);
        }
    }

    # Forward message
    while (1) {
        my $msg = zmq_msg_init();
        my $data;

        zmq_msg_recv($msg, $fromSocket);
        $data = zmq_msg_data($msg);

        if (zmq_getsockopt($fromSocket, ZMQ_RCVMORE)) {
            print "sending $data and more\n"  if ($self->{debug} > 2);
            zmq_send($toSocket, $data, -1, ZMQ_SNDMORE);
        } else {
            print "sending $data\n" if ($self->{debug} > 2);
            zmq_send($toSocket, $data, -1);
            last;
        }
    }
}

sub sendReady {
    my $self = shift;

    debug "Sending READY", $self->{debug};

    $self->{lastUpstream} = time();

    zmq_send($self->{upstreamSocket}, '', 0, ZMQ_SNDMORE);
    zmq_send($self->{upstreamSocket}, ZRTG_READY, -1);
}

1;
```

ZRTG::WorkerBee

```
package ZRTG::WorkerBee;

use 5.10.0;

use threads;

use strict;
use warnings;

# Various necessary libraries
use Log::Message::Simple qw(:STD);
use Time::HiRes qw(gettimeofday tv_interval);
use JSON;

# Should use a different SNMP module since the maximum message size
# with Net::SNMP will likely be problematic
use Net::SNMP;

# ZMQ Libraries
use ZMQ::LibZMQ3;
use ZMQ::Constants qw(:all);

# ZRTG Libraries
use ZRTG::Constants qw(:all);
use ZRTG::Payload;

# Thread entry point
sub main {
    my $context = shift;
    my $debug = shift;

    # Socket to talk to dispatcher
    my $dealer = zmq_socket($context, ZMQ_REP);

    if (0 != zmq_connect( $dealer, 'inproc://workers' )) {
        die "Could not connect dealer socket: $!";
    }

    while (1) {
        # Block while waiting for some data
        my $payload = ZRTG::Payload->new($dealer);

        # Handle incoming data
        if ($payload->{type} eq ZRTG_SNMP_SMART_REQUEST) {
            snmpSmartRequest($payload, $dealer, $debug);
        } elsif ($payload->{type} eq ZRTG_REFRESH_INTERFACES) {
            snmpGrabInterfaces($payload, $dealer, $debug);
        } else {
            # Probably don't really want to die here, but for debugging it's not
            # very nice if we hit this and things go into an infinite loop or
            # something
            die "Unknown request"
        }

        # Should flush log buffer periodically
    }
}
```

```
# Bulk walk an entire OID tree based on the number of interfaces
sub snmpSmartRequest {
    my $payload = shift;
    my $socket = shift;
    my $debug = shift;

    # Suck in arguments in the order in which we expect them
    my @oids = @{$payload->{data}};
    my $host = shift(@oids);
    my $community = shift(@oids);
    my $version = shift(@oids);

    # Log which thread this came from, helps out with debugging
    my $tid = threads->tid;

    debug "Processing SNMP_SMART_REQUEST for $host ($tid)", $debug;

    # Initialize an SNMP session
    my ($session, $error) = Net::SNMP->session(
        -hostname  => $host,
        -community => $community,
        -version   => $version,
    );

    # Store retrieved counter values here
    my @workerResults;

    if (! defined $session) {
        error "Could not create SNMP session: $error";
        return;
    }

    # Grab the number of interfaces on this device and record how long it takes
    my $t0 = [gettimeofday];
    my $snmp_interfaces = $session->get_request(-varbindlist => [ NUM_INTERFACES_OID ]);
    my $t1 = [gettimeofday];

    # We're going to say that's a pretty good indication of the RTT for this
    # device
    my $latency = pack('f<', tv_interval($t0, $t1));

    # If we got that back, let's go ahead and do bulk requests for all of our
    # OIDs using that count as our bulk limit
    if (defined($snmp_interfaces->{&NUM_INTERFACES_OID})) {
        my $count = $snmp_interfaces->{&NUM_INTERFACES_OID};

        # These next requests could get big...
        $session->max_msg_size(65535);

        # Theoretically interface name -> oid associations can change per-reboot
        # since the name is what we generally use for a key, grab that every time
        my $names = $session->get_bulk_request(-maxrepetitions => $count,
                                               -varbindlist => [ INTERFACE_NAMES_OID ]
                                              );

        # Now bulk walk the OID tree for each requested MIB, usingthe number of
        # interfaces for a max repetitions value
        for my $mib (@oids) {
            my $res = $session->get_bulk_request(-maxrepetitions => $count,
```

```
                                                -varbindlist => [ $mib ]
                                               );
            my $t2 = [gettimeofday];
            my $types = $session->var_bind_types();

            # Populate results hash based off of the time on this machine
            for my $oid (keys(%{$res})) {
                # oid
                # name
                # counter
                # time
                # type
                if ($oid =~ /^(.*)(\.\d+)$/) {
                    if ($1 ne $mib) {
                        debug "Dropping $oid since we didn't ask for it", ($debug > 1);
                        next;
                    } elsif (! $names->{INTERFACE_NAMES_OID . $2}) {
                        debug "Dropping $oid since we couldn't resolve an interface
name", ($debug > 1);
                        next;
                    }

                    push @workerResults, [ $oid, $names->{INTERFACE_NAMES_OID . $2},
$res->{$oid}, $t2->[0], $types->{$oid} ];
                } else {
                    warn "Uh oh, that's a weird looking oid: $oid";
                }
            }
        }
    } else {
        warn "Failed to find the number of interfaces on this device"
    }

    # And close the SNMP connection
    $session->close();

    debug "Shipping off result for $host ($tid)", $debug;

    my $out = ZRTG::Payload->new(ZRTG_SNMP_RESPONSE);

    $out->setData(
                    $host,
                    $latency,
                    encode_json(\@workerResults)
                );

    $out->send($socket, '');
}

sub snmpGrabInterfaces {
    my $payload = shift;
    my $socket = shift;
    my $debug = shift;

    # Suck in arguments in the order in which we expect them
    my $host = $payload->{data}->[0];
    my $community = $payload->{data}->[1];
    my $version = $payload->{data}->[2];

    # Log which thread this came from, helps out with debugging
```

```
    my $tid = threads->tid;

    # Store retrieved values here
    my %workerResults;

    debug "Processing REFRESH_INTERFACES for $host ($tid)", $debug;

    my ($session, $error) = Net::SNMP->session(
        -hostname  => $payload->{data}->[0],
        -community => $payload->{data}->[1],
        -version   => $payload->{data}->[2],
    );

    my $snmp_interfaces = $session->get_request(-varbindlist => [ NUM_INTERFACES_OID ]);

    if (defined($snmp_interfaces->{&NUM_INTERFACES_OID})) {
        my $count = $snmp_interfaces->{&NUM_INTERFACES_OID};

        # These could get big...
        $session->max_msg_size(65535);

        my $names = $session->get_bulk_request(-maxrepetitions => $count,
                                               -varbindlist => [ INTERFACE_NAMES_OID ]
                                              );

        my $aliases = $session->get_bulk_request(-maxrepetitions => $count,
                                                 -varbindlist => [ INTERFACE_ALIASES_OID
]
                                                );

        my $speeds = $session->get_bulk_request(-maxrepetitions => $count,
                                                -varbindlist => [ INTERFACE_HIGHSPEED_OID
]
                                               );
        my $adminStatus = $session->get_bulk_request(-maxrepetitions => $count,
                                                     -varbindlist => [
INTERFACE_ADMIN_STATUS_OID ]
                                                    );
        my $operStatus = $session->get_bulk_request(-maxrepetitions => $count,
                                                    -varbindlist => [
INTERFACE_OPER_STATUS_OID ]
                                                   );
        $session->close();

        for (keys(%{$names})) {
            /^(.*)\.(\d+)$/;
            if ($1 ne INTERFACE_NAMES_OID) {
                debug "Dropping $1 since we didn't ask for it", $debug;
                next;
            }
            $workerResults{$2}->[0] = $names->{$_};
        }

        for (keys(%{$aliases})) {
            /^(.*)\.(\d+)$/;
            if ($1 ne INTERFACE_ALIASES_OID) {
                debug "Dropping $1 since we didn't ask for it", $debug;
                next;
            }
            $workerResults{$2}->[1] = $aliases->{$_};
```

```
        }

        for (keys(%{$speeds})) {
            /^(.*)\.(\d+)$/;
            if ($1 ne INTERFACE_HIGHSPEED_OID) {
                debug "Dropping $1 since we didn't ask for it", $debug;
                next;
            }
            $workerResults{$2}->[2] = $speeds->{$_};
        }

        for (keys(%{$adminStatus})) {
            /^(.*)\.(\d+)$/;
            if ($1 ne INTERFACE_ADMIN_STATUS_OID) {
                debug "Dropping $1 since we didn't ask for it", $debug;
                next;
            }
            $workerResults{$2}->[3] = $adminStatus->{$_};
        }

        for (keys(%{$operStatus})) {
            /^(.*)\.(\d+)$/;
            if ($1 ne INTERFACE_OPER_STATUS_OID) {
                debug "Dropping $1 since we didn't ask for it", $debug;
                next;
            }
            $workerResults{$2}->[4] = $operStatus->{$_};
        }

        debug "Shipping off result for $host ($tid)", $debug;

        my $out = ZRTG::Payload->new(ZRTG_INTERFACE_RESPONSE);

        $out->setData($host,
                      encode_json(\%workerResults)
                     );

        $out->send($socket, '');

    }
}

1;
```

Appendix C – Example execution of ZRTG