Hosting a beacon

Authors: Ulrich Speidel (ulrich@cs.auckland.ac.nz), ‘Etuate Cocker (ecoc005@aucklanduni.ac.nz)

Date: 19 September 2013  -- THIS DOCUMENT IS A DRAFT --

Abstract
This document is intended for anyone considering hosting a beacon on their organization’s network. It describes the technical and infrastructure requirements.

What is a “beacon”?
The purpose of our network is to measure and investigate long-term trends in the reliability, timeliness and consistency of packet and real-time content delivery in traffic such as Voice over IP (VoIP) in a global setting. Ultimately, this should assist us in finding an answer to the question whether the Internet is headed for ultimate stability or for chaos, whether the future availability of today’s services is in jeopardy, or whether we can count on them becoming increasingly reliable.

For this purpose, we’re building a global network of “beacon” computers to study the long-term trends in these areas. A beacon is simply a computer running the beacon software we have developed.

Each beacon conducts at most one experiment at a given time, in cooperation with one of the other beacons on the network. In the experiments, the two beacons exchange synthesized data that allows the detection of packet loss and the recovery of timing information. Both the transmitted and received packets are logged and these logs are archived. Most experiments are repeated a few times per day on an ongoing daily basis (very much like someone would measure and log temperature or pressure at a weather station). I’ll explain a bit more about the current set of experiments below, but a typical experiment simulates the kind of traffic two computers would exchange during a three minute voice over IP (VoIP) telephone call, e.g., using Skype.

As measures, we will use jitter, packet loss, path hop length and various entropies and complexities of the received packet stream. Since all packet transmission and arrival timing, hop information and packet losses are logged, anyone can apply whichever measures they would like, too, in retrospect – so your ideas are welcome.

Selecting a machine
Hardware
Ideally, a beacon should run on a dedicated machine that does nothing else. The reason for this is that any other processes running on the machine take CPU time and may influence the accuracy of the beacon’s transmit and receive timing. That said, any machine that is reliably able to tend to the beacon
process frequently enough so the beacon’s main loop can check for packets many times per milliseconds is acceptable — and this, in practice, include most computers. Machines to avoid include compute servers running computing-intensive tasks such as simulations, graphics generation, games or anything else that you would try to be able to compute faster if you could. File, web and e-mail servers subject to heavy load are also not suitable, although in practice, these may sometimes be the only machines available. There is a relatively simple test to establish whether a beacon is straining under load from other applications.

The beacon process itself is time-critical, but not computing –intensive, so almost any modern computer has sufficient processing power to host a beacon. We use, among others, mini PCs based on Alix boards with 16GB CF cards as memory / disk. The machine does need a real-time clock, though, so a plain Raspberry Pi is not enough!

Software and operating system
The original software platform is Ubuntu Linux. We also have beacons running at a number of sites using FreeBSD and OS X, and are currently testing Voyager as a lightweight alternative to Ubuntu. Ultimately, whether the software will run on a particular platform depends on the data structures available in the networking libraries on the respective platform.

The beacon is able to use the sendemail utility to send notifications to (by default ‘Etuate Cocker, but this is configurable) to alert us to unreachable beacons (TCP experiments only) that may be down. Under Ubuntu, this utility is installed via:

```bash
sudo apt-get install sendemail
sudo apt-get install libio-socket-ssl-perl
```

One important criterion in the OS category is that the OS must be context switching sufficiently frequently to permit near real-time timestamping of packet events. Operating systems which are not configured to do this deliver high baseline jitters.

Location and network
We are often asked whether we want a machine “close to the backbone” or “in the data center”. While that is OK and often offers advantages in terms of 24/7 operation, reliability of power supply, and so on, it’s not a requirement: Normal users make Skype calls from their desktop, and whatever their packets encounter is OK for ours to put up with. Since the machine is meant to operate for a long time, it should be stationary, however – it’s best if you don’t volunteer a laptop that you will unplug twice daily and operate from different locations. Similarly, it’s also best to choose a machine that third parties are not likely to interfere with – so a machine in a student lab that will be re-imaged by the next incoming graduate student is also not a good choice.

The machine should have a fixed IP address – dynamic DNS is an option but not at all preferred. An IP address can be changed (e.g., as part of a network restructuring exercise), but this may require all corresponding beacons to be reconfigured. So preferably, this shouldn’t happen too often.
Reliability and longevity
This is a longitudinal study, and short-term outages are of no real concern. If you were to plot long-term temperature trends for Tonga, and the thermometer broke down for two weeks in December, it’d still safe to assume that the period didn’t contain a mini ice age. In other words, you don’t need a dual power supply, RAID array etc.

Wherever possible, select a machine / location that is likely to be available long-term (or likely to be able to be replaced as it reaches the end of its life). It is not so important that the network topology around the machine stays stable – a part of what we are interested in is the influence that changing network topology around client machines has, as this seems to be an integral part of network evolution over time.

Time base
In order to keep the experiment accessible to a large number of sites, we do not require beacons to be synchronised to high precision by an atomic clock or GPS clock. This means that absolute latencies measured will always include an error equal to the clock offset between the two beacons involved. However, when computing jitter as the mean variation of the latencies, this error is irrelevant as long as it can be assumed to be constant over the duration of a single experiment (typically a few minutes). To achieve this, the beacon’s real-time clock should not drift by more than a few seconds a day. In order to schedule beacon experiments, we also require beacons to synchronise hourly with a network time protocol (NTP) server.

Access, security and privacy
This is a question that comes up a lot. ISPs and university computer centers tend to feel a little uneasy to host an Internet application that communicates with machines outside the university. This section discusses some of the issues we have encountered to date (but there will doubtless be others).

- “How do I know what your software does?” – Answer: We’ll give you the source and you can have a look for yourself, and compile it for yourself. The code is heavily biased towards speed, so you’ll find relatively more code repetition and fewer function calls than in “normal” code – this is intentional.
- “Why do you need root, sudo, or setuid privileged level access and why do you need to run a raw socket?” – Answer: Most operating systems require elevated access privileges to generate a raw socket, because a raw socket lets the application using it create arbitrary packets and listen in on any traffic – ideal if someone would like to eavesdrop or craft packets for an attack. We need a raw socket because it is the only way to access the header of incoming IP packets, which we need to read the value of their time-to-live fields (TTL). The value of this field tells us how many routers a packet has passed through on the way to the receiver. A change in this value is a “smoking gun” proof that the path between transmitter and receiver has changed. We don’t use the raw socket to read other traffic than our own packets, and we certainly don’t use it to generate any packets. Also: You don’t need to give us any of these privileges, as long as someone in your institution with these privileges is able to compile, install, schedule and run the beacon program for you.
• “Why does the machine need to be open for UDP traffic?” – Answer: UDP is the most common protocol used by Voice over IP (VoIP) applications such as Skype. Sure, some of these applications also work with TCP if they have to. However, we are interested in the long-term viability of VoIP, and so would like to know how reliable the arrival timing for VoIP packet streams is. This means that the machine has to be able to exchange UDP packets with other beacons. N.B.: We also monitor the viability of VoIP over TCP channels, so the machine needs TCP access as well.

• “Which packets do you monitor?” – Answer: The beacon software will “see” every packet present at the network interface and will check its destination address and port for a match with its own address and the port used for the respective experiment. No further information from the packet is inspected unless there is a match. Only information from matching packets (i.e., packets transmitted by other beacons as part of the experiments) is logged. N.B.: At some of our beacon sites, we also run netflow/NTOP/NPROBE traffic monitoring to ascertain Internet usage patterns – this is a separate experiment, however, and is not part of the beacon functionality.

• “Do the beacons transmit any personal information or make any personal information accessible?” – Answer: This question came from our Canadian site at the University of Victoria, and has also been posed by another prospective site in Canada, as privacy legislation there obliges universities to place tight controls on what information can and cannot leave the university. It’s quite possible that this may also apply elsewhere. As already explained above, we don’t log information that does not come from other beacons. Moreover, the only information we transmit is comprised of: packet serial numbers, information identifying the experiment and experiment run, the beacons involved, their addresses and ports, as well as timestamps, TTL values, and similar non-personal information. The data is augmented with stuffing from a file supplied – this contains contact information for ‘Etuate and the note that it’s a beacon experiment, but can be replaced with your own dummy data of choice. When a TCP experiment fails because a connection could not be established, the beacon also sends e-mail notifications (to ‘Etuate by default, but this is configurable). These are simply one-liners indicating which remote beacon your beacon has been unable to connect to.

• “Will you run a server, i.e., will there be ports open on the machine?” – Answer: Yes, but not all the time. The beacon software does not run continuously but is started as a cron job at the time of the experiments, typically for a few minutes each. Ports close again after the end of each experiment.

• “Which ports need to be open on the firewall?” – Answer: we typically use port 8088 for both TCP or UDP, but this is configurable if we need to bypass firewalls that are not under our or the beacon host’s control (e.g., we also use ports 12000 and 5088 at some sites). Also, in order to be able to retrieve the log files, we need to be able to rsync. If we are meant to configure and/or schedule the beacon operation, we also need ssh access and an appropriate privilege level (root / superuser) on the beacon machine. Finally, the beacon generates a notification e-mail if it hasn’t been able to connect to a particular partner beacon. This requires the installation of sendemail on the host machine and the opening of port 995 for outgoing connections to smtp.gmail.com. Unless the machine has an alternative time server configured, port 123 UDP
will also need to be opened to allow the beacon to synchronise its clock with an NTP server on the Internet. This ensures that the beacon activates at the right time and does not miss its scheduled communication with its remote partners.

- “Which types of traffic will the beacon generate?” – Answer: This depends a little on which experiments are configured, and also on which other beacons your beacon will be configured to communicate with. The UDP experiments we run have either uni-directional or bi-directional flow and mimic either Skype voice audio or Skype video packets. UDP “voice” packets have a default payload of 115 bytes each, those for “video” use 300 bytes. If you inspect the content of one of these packets, you will find (a) a packet serial number, starting at 0 in each experiment run, (b) the transmit timestamp inserted by the sender, (c) an experiment number, identifying the experiment that this transmission is part of, and (d) a run number, identifying the instance of the experiment (every time the experiment is carried out between two beacons, it gets a new run number). The rest of the data in the packet is stuffing from the file beacon.txt, which by default contains the text “BEACON PROJECT TEST PACKET PLEASE IGNORE - CONTACT ecoc005@auckland.ac.nz FOR MORE INFORMATION.” However, you can configure your own file if you like. The stuffing is not recorded by the receiving beacon. In the case of TCP experiments, payload data is transmitted either uni-directionally or bi-directionally. The TCP experiments operate in two modes: “VoIP” mode, where data is sent to the transmitting socket in short snippets every 20 ms, and “download” mode, where 1 MB of data are sent to the transmitting socket as quickly as the socket is able to dispatch it. Finally, when running TCP experiments, you may occasionally see e-mail traffic to smtp.google.com, sending notifications to ecoc005@aucklanduni.ac.nz. These are sent whenever the beacon is unable to connect to its remote counterpart. If you’d rather have the notification sent to a local person, it’s easy to reconfigure the respective settings via command line options on the beacon, or you can suppress the notifications altogether.

- “We’re on a network with extremely limited bandwidth and are worried about the amount of traffic your experiments may cause. Can we still take part?” – Answer: This is a problem we face in many Pacific Island locations. We can accommodate by: (a) not scheduling “video” type UDP or TCP experiments or “download” type TCP experiments, (b) not scheduling experiments at times when your links are really busy, and (c) scheduling experiments with fewer remote beacons.

Please ask us if you’d like to know more or have questions we haven’t addressed above.

**Compiling the beacon software**

The beacon software itself consists of the file `beacon_soft.c`, the main code file that contains the UDP and TCP code for the beacon, the logging code and some auxiliary functions. You’ll notice that it isn’t overly structured in the sense that most of the code resides within the main() function. This is deliberate as we’ve wanted to avoid time-consuming function calls wherever possible – accurate timing is more important!
Compile with:

gcc beacon_soft.c -lm -o beacon

Installation

Copy or move the beacon file to its final location. We use two copies, one in /usr/Experiments/UDP and one in /usr/Experiments/TCP. This allows us to migrate to new versions without the risk of interrupting experiments too much, e.g., if we make amendments to UDP-related code, we can road-test in /usr/Experiments/UDP first and only update in /usr/Experiments/TCP once we’re happy with the result.

Each directory that you install the beacon in also needs the following:

- A file beacon.txt which contains data that the beacon “stuffs” (pads) its packets and other transmissions with. Most packets that the beacon sends need to be longer than the data the packet needs for serial number, timestamp etc., and beacon.txt provides additional text or bytes that the beacon uses to fill the remaining space.
- A subdirectory named logs. The logs subdirectory stores the logs and needs to be writeable by the beacon user.
- A subdirectory named rid. The rid subdirectory stores the run ids for each experiment and needs to be readable and writeable by the beacon user.

The machine that the beacon runs on must also synchronise its real time clock at least once a day via NTP (Network Time Protocol) with a time server.

Configuring an experiment

Before you schedule experiments, you’ll need to configure them. This is done exclusively via command line options:

    beacon -h

or

    beacon -?

gives you a list of command line options available. Normally, you will configure an experiment in conjunction with the beacon that will run the remote end of the experiment. To do so, the modes in which the beacons need to operate (command line option -m) need to match:

<table>
<thead>
<tr>
<th>Local beacon</th>
<th>Remote beacon</th>
</tr>
</thead>
<tbody>
<tr>
<td>-m 0</td>
<td>-m 1</td>
</tr>
<tr>
<td>Transmits UDP, specifies experiment number, must start after remote beacon</td>
<td>Receives UDP, receives experiment number and run number from far end (local beacon), must start first</td>
</tr>
<tr>
<td>-m 1</td>
<td>-m 0</td>
</tr>
<tr>
<td>Receives UDP, receives</td>
<td>Transmits UDP, specifies experiment number,</td>
</tr>
</tbody>
</table>
Beacons running in even-numbered modes are called “initiators”, beacons running in odd-numbered modes are called “responders”. In simple terms, you may think of them as clients and servers, respectively, except that each beacon process runs only for a short time for the purpose of a single experiment run. For this reason, responder processes must be up and running before the initiator starts, so the packets sent by the initiator have a process at the far end that can receive them.

**Experiment and run numbers**

The longitudinal nature of the beacon experiments means that the same experiment is repeated between different beacons on a regular basis. Each experiment is uniquely identified by a number (which is the same across all beacons that run this experiment). On each beacon, each repeat of the experiment (no matter which partner beacon is used) is given a run number and a unique start timestamp.

Example: Beacon EX1 runs experiment 42 as initiator with partner beacons EX2 and EX3. The first time EX1 runs the experiment, it communicates with EX2. This will be run number 1. The next time, it communicates with EX3. This will be run number 2. The next time, it then communicates with EX2 again.
and this will be run number 3. EX1 stores the respective current run number in the rid directory in a file called experiment42.rid. [Note: Due to past fsck-related write permission issues on some of our solid state memory-based beacons, these run ID files were not always updated correctly and as a result, experiment/run number combinations in our existing data are not necessarily unique. However, identical combinations can still be distinguished chronologically by their time stamps.]

**Basic functioning**

After starting beacon, the beacon software reads its command line configuration. If the beacon is to run as an initiator (even modes), the beacon determines the run number to be used from the corresponding .rid file in the rid directory.

It then creates:

- A UDP socket (in modes 0 to 3)
- A TCP socket (in modes 4 to 7)
- A raw socket whenever the mode involves receiving data (in modes 1, 2, 3, 5, 6, and 7)

In modes 1, 2, 3, 5, and 7, the socket is bound to the specified ports. All these sockets are non-blocking, meaning that the beacon periodically checks them for pending packets that have been received, rather than halting beacon execution while waiting for incoming traffic.

In modes 4 and 6, the beacon connects to its remote counterpart via TCP and communicates the experiment and run number to the remote beacon.

The beacon then enters a loop, which it remains in until:

- The specified number of UDP packets have been transmitted (mode 0)
- The specified number of seconds has elapsed (mode 1)
- The specified number of UDP packets have been transmitted and the specified number of seconds has elapsed (mode 2 and 3)
- All data has been transmitted and the TCP connection can be disconnected, or until timeout is reached (mode 4)
- All data has been received and the TCP connection has been disconnected, or until timeout is reached (mode 5)
- All data has been exchanged and the TCP connection has been disconnected, or until timeout is reached (mode 6 and 7)

In UDP experiments that receive, the loop first checks whether a packet for the port has been received from the counterpart beacon on the raw socket. If so, it is timestamped and its TTL value extracted. It is then formally received via the UDP socket later in the loop (the timestamp is contributed by whichever socket sees the packet first). In UDP experiments that transmit, the beacon checks in the loop as to whether the specified time period since the last packet transmission has elapsed and transmits the next packet where appropriate.
In TCP experiments, there is a similar loop. The main difference is that TCP experiments transmit chunks of data rather than packets – the assembly and management of the packets themselves is left to the TCP/IP stack. In VoIP mode, data chunks are transmitted at regular intervals (every 20 ms by default). Download mode pushes data into the connection as quickly as possible.

After the end of the respective experiments, the beacon writes the corresponding log files and shuts down.

In UDP experiments that transmit maintain a loop iteration counter. Its value is recorded in the T log file for each packet transmission and is reset to 0 after each transmission. The counter value in the log tells us how many times the loop was able to iterate between packet transmissions. This is useful as it gives an indication of the number of iterations per millisecond and hence of the accuracy of relative timestamps. Based on a 20 ms inter-transmission interval, a counter value of 1000 indicates 50 iterations per ms, or around 20 microseconds intervals between packet detection and timestamping opportunities. Ideally, these intervals should not exceed 100 microseconds, so log files in which counter values of under 200 are observed indicate an overloaded system (note that the first value is always 1).

**Log files**

Each time a beacon runs an experiment, one or more log files are generated. The names of these log files are composed as follows:

1. Three letter code of the local beacon (the beacon that is writing the log file), followed by an underscore (“_”)
2. Three letter code of the remote beacon involved, followed by an underscore
3. Experiment number, followed by an underscore
4. Run number, followed by an underscore
5. Timestamp in seconds since 1 January 1970 (“UNIX timestamp”), followed by an underscore
6. “T” for transmit log or “R” for receive log, followed by an underscore
7. Mode (as per –m flag on the beacon writing the log), followed by an underscore
8. “UDP” in case of UDP experiments, or “TCP” or “RAW” in case of TCP experiments.
9. The extension “.txt”

Example: NZ1_CK1_3_59_1339560061_T_0_UDP.txt is a UDP transmit log for mode 0 written by beacon NZ1 having transmitted to beacon CK1 as part of experiment 3. It is the 59th time that this experiment has been run on beacon NZ1, and the beacon started 1339560061 seconds after midnight 1 January 1970 according to NZ1’s clock.

Note that the time stamp always reflects local beacon time, and will be different on the receive log that corresponds to the transmit log. E.g., the receive log file name could be CK1_NZ1_3_59_1339560001_R_1_UDP.txt, reflecting the fact that the responder beacon (which runs mode 1) was started a minute before the initiator. Corresponding logs are identified via time proximity, not exact time match.
Each beacon log file contains a header which records various other information, including the beacon configuration.

One feature supported by the latest version of the beacon software is a loop iteration count.

**Scheduling**
Finally, you’ll need to schedule the experiments via a cron job – this usually needs coordination with the administrator of the corresponding remote beacon. Typically, we have scheduled most experiments to run at 8 hour intervals for each beacon pair, i.e., to have three runs per day between the same destinations.

By default, the sockets in modes that receive listen for 240 seconds to allow for clock differences and latencies of up to 20 seconds between the machines (the default duration of an experiment is 200 seconds). With frequent NTP synchronisation, it should not be necessary to start the responder more than about 2 seconds before the initiator, however, on beacons in low-bandwidth locations, we typically schedule each beacon with a listening time of 300 seconds (\(-1\ 300\)) and allow for an earlier start of the responder.

**Log file retrieval**
In order to have the log files retrieved to our central repository, we typically rsynch them once a day. ‘Etuate is happy to discuss the details.

**Analysis**
We have (some) code to assist us in the analysis of log files, which we are happy to share with any research group involved in the beacon project. We are also able to give you access to the existing [large] beacon log file data set. Much of this is now downloadable in the form of monthly zip files via our plotting web interface – ask us for access.