Introduction

This is a draft copy (so there will be spelling and grammar mistakes) of a high availability system specification by Richard Bairwell of Bairwell Ltd - https://www.bairwell.com/ - and is not a final document and a large number of assumptions have been made, no actual coding has taken place for “proof of concepts”, reliance on unknown third party internet sources have been made, and therefore this document should be taken as a “jumping off” discussion point rather than a considered/in-depth final analysis of the problem.

Problem as stated

> Use the PDF Product Specification for the Identicom.
>
> This product can communicate via SMS and GPRS, for the purposes of your analysis please only focus on the GPRS communication.
>
> Assumptions:
>
> > 10,000 Units
> >
> > Each communicating every 30 minutes
>
> Requirements:
>
> Outline in a brief document the application you would design to manage the incoming messages.
>
> Which technologies/languages you would use.
>
> Please include a basic flow diagram/network diagram.
>
> System Capacity (This depends on the hardware its running on but basic assumptions are fine)

The majority of this is covered in the “Outline” section with a network diagram later on (under “Managing the messages”). Justification is provided for the choices outlined in later sections, but this is the result of just 4 hours investigation, research and writing and therefore there could be missed solutions.

Outline

To handle the necessary requirements, I would have a fully load balanced solution (as, for safety purposes, we cannot afford to “lose messages”), with a message broker dealing with how messages are actually handled and a number of “receiving stations” which would act as the “frontend” between the Identicoms and the rest of the system.

The maximum capacity of the system, from the message broker onwards, is theoretically unlimited as messages will just be queued up and new servers can be deployed when necessary. The capacity constraints will come from the front load balancers, the message broker and finally the “receiving stations”.

The receiving stations need to be responsive, reliable and fast and therefore the programming language(s) C/C++ would be ideal – but a decision will need to be made whether the extended development time needed for a safe, secure, reliable C/C++ program would over-rule that of a more “pre-built” solution such as Go or PHP: which would require more hardware resources.

Working on the following assumptions:

- Dual HAProxy load-balancers (max capacity expected 25,000 rps)
- Dual high availability C++ receiving stations (max capacity expected 17,000 rps)
- Redis on a Xeon e5520 as a message broker (max capacity expected circa 500,000 rps)
- Single PHP/MySQL logging solution (max capacity about 100 rps, but is behind the message broker)
- Single PHP/MySQL high-priority alert solution (same max capacity). Assume only 1 in 100 messages is a high-priority. Limit would therefore be 10,000 rps.
- Single PHP/NFS media storage solution (will be around 50 rps due to file size/replication at a guess, but again is behind the message broker). Assume only 1 in 100 messages is a photo. Limit would therefore be 5,000 rps.

We could then have a theoretical receive capacity of 17,000 rps, but for rapid processing the limit would be 100 rps. Over an hour (baring in mind high-priority messages will be specially treated), the maximum capacity would be 360,000 messages.

Identicom communications/Needs Statements

Assume 10,000 units are in operation (over GPRS/TCP/IP only) and communicate with the servers every 30 minutes.

As these messages can be "life critical" (in the instance of "Man down" style communications), it is vital messages do not get "lost".

The protocol of the device, as transmitted and received over TCP/IP is as follows:

Position	Bytes	Field	Notes
0	1	Frame start	0x1e (fixed)
1	1	Type	Message definition
2	0...	Data	As defined
?	1	Frame end	0x1e

Multi-byte fields are presented with least-significant byte first, signed numbers are 2-complement.

If a control character needs to be sent it is transmitted as an escape code 0x1d + original data 0x40 - i.e. 0x1e is sent as 0x1d 0x5e and 0x1d is sent as 0x1d 0x5d.

The Identicom has a maximum receive data size of 120 characters (excluding “byte-stuffed”/”escaped” characters – so theoretical maximum of an entire message is 3+ (frame start, type, frame end) and 120x2 = 243.

The Identicom appears to be able to transmit messages with a data size of up to 1024 (excluding “byte-stuffed”/”escape” characters) - so 2048+3 = 2051 characters.

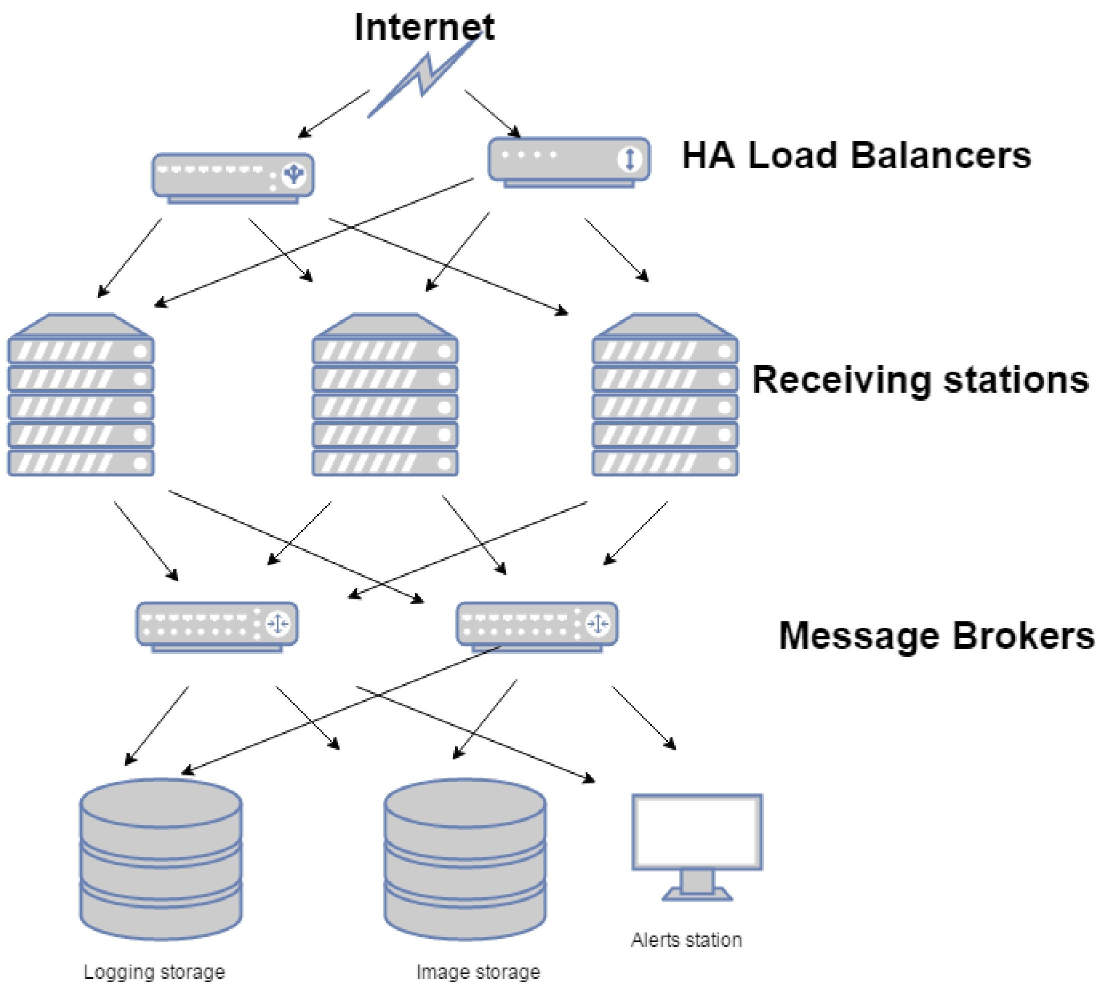
As the Identicom connects to the server at set intervals (or when an “event” is triggered), the server cannot rely on the Identicom always being available – when the Identicom “phones home”, all necessary commands must be provided by the server: this will require appropriate “logic” in the receiving server [ruling out a simple “dumb” receiver solution].

Certain messages (such as reading pictures - message code 0xc0-0xc3) need to be transmitted/received sequentially/repeated for a single “transmission” to be completed.

Working on this assumption, we can expect (if each Identicom were to connect and send the maximum allowed amount of data), 10,000* 2051 =19.6Mb of data (the majority of the messages will be shorter, but some data blocks may actually require multiple messages to be sent). This is an expected “worse case scenario peak”.

Managing the messages

We will need the following high-availability redundant infrastructure in an “ideal world” situation:



This is detailed further in the following pages.

2 or more TCP/IP loader balancers

To handle failure of the “receiving” stations (a redundant pair should be installed in case of a failure of a single load balancer). These must operate on a strict TCP layer 4 [transport] as the data format communicated is proprietary and “web load balancers” (which operate on layer 7 [application]) will try and interpret the incoming data as HTTP or other protocol specific data – leading to corruption.

Example Solutions

Amazon AWS Classic Load Balancer (Layer 4)[1]

Custom hardware solutions (either off the shelf of ASIC custom solutions)

HAProxy[2] - however, the throughput of a software based solution (such as HAProxy) will be a lot lower than that of a dedicated hardware device.

The Rackspace Cloud Load Balancer does NOT look suitable (as it appears to be protocol-specific layer 7).

Network Security

These should be in the DMZ area of the network.

Capacity

“Amazon AWS Classic Load Balancers...do not cap the number of connections[3]”

Barracuda 240 Load Balancers[4] have a maximum throughput of 100Mbps - which based on the circa 2Kb maximum per message from the Identicom means it should handle 51,200 active Identicoms all reporting at the same time.

HAProxy[5] can handle circa 25,000 requests per second - so it should cope with the “worst case scenario” of 10k.

1 https://aws.amazon.com/elasticloadbalancing/classicloadbalancer/
2 http://www.serverlab.ca/tutorials/linux/network-services/layer-4-load-balancing-with-haproxy/
3 https://aws.amazon.com/elasticloadbalancing/classicloadbalancer/faqs/
4 https://www.barracuda.com/products/loadbalancer/models
5 http://www.goperf.com/haproxy-experimental-evaluation-of-the-performance/

2 or more “receiving” stations

These are behind the load balancers (to cover failure states) and handle the initial connection/communication with Identicom units. These need to be reliable, fast but also understand the Identicom protocol and make further requests as/when necessary.

These receiving stations will also filter inappropriate messages (such as invalid commands/port scans) and also should filter DDoS style attacks.

The receiving stations are the only part of the infrastructure (with the exception of the load balancers) which communicate directly with the Identicom and is the only part of the system expected to have capacity issues (as once the messages are on one or more message brokers, the individual services can pick the messages up “as and when”).

Whilst it does not appear documented in the specification, and therefore not covered here, there may be a need to hold things such as Firmware Updates for the Identicoms on the receiving stations and a mechanism for them to be provided to them. This should be bared in mind for future expansion.

Requirements

The receiving stations will need to:

1. Receive TCP/IP packets from the internet (hopefully from an Identicom!)
2. Unpack the data based on the frame data

   Whilst labelling this as “Unpacking”, it just means checking the first and last bytes are 0x1e, the 2nd byte is a recognised type and then taking the remaining bytes, unescaping/byte-de-stuffing them, and then processing them as per the type.
3. Take action based on the recognised type

   If this is the first message in the connection, then validate it is an expected Identicom.
4. Repeat step 2 and 3 until all event log messages/photos are downloaded.
5. Issue a 0x03 (delete event log) message.

As the receiving station needs to be as quick as possible (to ensure the Identicoms are not needlessly connected and transmitting wearing down battery live), which ever system we use should be run a server daemon/long-running process to prevent the “startup” times of applications (which may only be 500ms, combine that with 30 minute check-ins on 10,000 units and that's 5.6 days worth of combined battery wasted).

Example Solutions

For speed, I would be tempted to use a C/C++ daemon application: however, this would lead to complexity and C/C++ does “give you enough rope to hang yourself with” (especially if it is “directly facing” the internet). This would lead to the highest possible throughput (and lowest memory/CPU utilisation), but may take longer development time. As other servers, such as nginx, Apache, haproxy and sendmail, are all written in C, we should be able to achieve their similar 17,000+ requests a second.[6]

I would also look at perhaps running PHP as a daemon/long-running process (with a watchdog to restart it if necessary) on multiple servers.

PHP (running under HHVM[7]) can support between 192 and 1739 transactions per second (depending on framework), so that would mean to cope with our “worse-case scenario”, we would need between 6 and 52 “servers” available to cope with the load.

Go[8] (aka golang) is another programming language which may be suitable - it appears to be, at least under simple test conditions, about 3 times the speed of PHP[9]: handling 4,500 requests per second. Since Go compiles to “near C”, it may be the best solution without getting to “close to bare metal C”.

Node.js could be another potential - it'll save on development time as many coders know aspects of Javascript - but it is slower than Go: sometimes only slightly, but other times quite large[10].

The final decision would be a trade-off in hardware cost and development time (PHP, whilst being slower but known to a large number of developers, would have a higher reoccurring hardware cost but lower development time: C++ would have the lowest hardware cost, but much higher/longer development time).

Network Security

These should be in the DMZ area of the network.

Capacity

With a PHP solution, 25 servers should cope with the “worse-case scenario” load (all 10,000 connections at once) as long as the code is “tight” and does the minimum necessary.

With a Go solution, we would need at least 4 servers to safely cover the load.

With a C/C++ solution, we might be able to run it on just two servers.

6 http://dan.hersam.com/2015/02/25/go-vs-node-vs-php-vs-hhvm-and-wordpress-benchmarks/
7 https://kinsta.com/blog/the-definitive-php-7-final-version-hhvm-benchmark/
8 https://en.wikipedia.org/wiki/Go_(programming_language)
9 http://dan.hersam.com/2015/02/25/go-vs-node-vs-php-vs-hhvm-and-wordpress-benchmarks/
10 https://jaxbot.me/articles/node-vs-go-2014

2 or more “receiving station databases”

To enable the receiving stations to validate Identicoms (when they transmit their serial number and software version), they need to have access to a list of recognised Identicom serial numbers. These databases can be “read only”/”slaves” as the “master” list of Identicoms can be held managed/elsewhere.

Since we only need to check “Is this a recognised Identicom serial/version” and the data requirements for this is 18 bytes (according to message type 0x81), this is a total storage capacity of 10,000 x 18 = 176Kb (plus any indexing, record terminating overhead).

There will be very few updates to this list – only when an Identicom is (de)commissioned or updated will a change be required.

Due to the low storage capacity required (and low update frequency), it is possible to host these “databases” on the receiving stations themselves or “hard-code” the list into the receiving station software (with the ability to update the list).

Example Solutions

MySQL 5.7 in read-only mode should manage in excess of 900,000 queries per second[11] - but that is on a 40-core hyperthreaded 2.5Ghz machine with 512Gb RAM[12] : a reasonably expensive setup!

Memcached on an 8-core low-RAM machine should cope : indeed, Facebook have it handling 200,000 requests per second[13].

Network Security

These should be in the “internal” area of the network.

Capacity

Over 200,000 requests per second.

11 https://www.mysql.com/why-mysql/benchmarks/
12 http://dimitrik.free.fr/blog/archives/2015/10/mysql-performance-yes-we-can-do-more-than-16m-qps-sql-on-mysql-57-ga.html
13 https://www.facebook.com/note.php?note_id=39391378919

High availability message broker

Different messages will require handling in the back-end in different ways- for example, “Man down” log entries (type 0x82, event type 0x3b - or lanyard-pin removed event 0x48) will require high priority actioning), whereas event such as status check (0x20), charger connected (0x30) can be practically “ignored” (save for the fact they should reset a “last check-in” date for the Identicom and if a check-in “misses” an expected window, an alert should be raised).

The message broker should operate on a “Pub/Sub” (publish/subscribe) method allowing multiple “consumers” of the different types of message to subscribe to the entries and allow replacement/introduction of new consumers as the business requirements change.

The message brokers need to be set-up in a high availability scenario as we cannot afford (due to the high-priority events) to lose messages. It is possibly advisable to have one “message broker” cluster for normal (and high-priority) notifications and another dedicated to high-priority notifications only – therefore giving totally independent routing for critical messages.

For large messages (such as images), it may be necessary to bypass the message broker due to size limitations in the broker.

Example Solutions

Amazon AWS Simple Notification System (SNS), RabbitMQ, Redis

Network Security

These should be in the “internal” area of the network.

Capacity

Amazon SNS “is designed to meet the needs of the largest and most demand applications...unlimited number of messages”[14].

Redis on a Xeon e5520 can handle in excess of 552,028 requests per second[15]

Google has pushed RabbitMQ to handle in excess of one million requests per second[16].

14 https://aws.amazon.com/sns/details/
15 http://redis.io/topics/benchmarks
16 https://blog.pivotal.io/pivotal/products/rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine

Handling Stations

The number and type of these will vary depending on how each message needs to be handled.

However, I suggest at least initially:

- A basic “logging” server (with associated database) holding the log messages received
- A “media storage” server (to hold received pictures)
- An “alerts” system which only deals with high-priority messages and takes appropriate action (SMS alerts to members of staff, on-screen notifications etc etc).
- The justification for the alerts system being separate from the logging system it is possible that all 10,000 units try communicating at the same time and if logs are being spooled to databases, there may be an unacceptable delay in dealing with high-priority messages.

Using a “pub/sub” message broker means that these sections can be scaled up/down as and when needed.

It may also be advisable to have an additional system just monitoring the status of the message broker (to ensure there is no backlog occurring).

Example Solutions

Logging - This could be a simple PHP/MySQL solution (either running PHP7 or HHVM). Whilst the throughput may not be massive (maximum of 1,739 transactions/second on Drupal 8 on HHVM[17]), this shouldn't be a bottle-neck as messages will just stack-up on the message broker until they are logged. Even if it could only deal with 100 messages/second (slower than Magento 2), it would still deal with all 10,000 messages in 1.5 minutes without any impact to the Identicom units.

Media Storage - This should be a shared storage solution (such as a gluster mounted drive or NFS) which files are just added to. There will need to be a way of indexing/searching for them (perhaps Identicom serial number and datestamp) and a way of reading from the message broker (PHP?). Again, since the message broker can queue messages, the throughput of this is reasonably irrelevant.

Alerts System - Due to the relatively low volume of high-priority messages, this again could just be a simple PHP/MySQL solution: as long as we don't get more than 100 high-priority messages a second, there should be no impact to the handling time.

Network Security
These should be in the “internal” area of the network.

Capacity
Over 360,000 messages an hour for the logging and alerts system each on their own single-server solution.

17 https://kinsta.com/blog/the-definitive-php-7-final-version-hhvm-benchmark/