ANACOM DVB-T Monitoring System

4 YEARS IN PRODUCTION

CENTRO 2020
PORTUGAL 2020
UNIÃO EUROPEIA
Fundos Europeus Estruturais e de Investimento

1. Introduction

At the time of the Portuguese national digital television transition from analogue television broadcasting to digital television (also referred to as **digital switchover**), Portugal's national regulator (ANACOM) was the target of a considerable amount of criticism by the general population as well as by consumer protection organisations such as DECO. This criticism had its origin in the failures and lack of quality of experience perceived by end users and associated with the new digital television service. To be able to provide the necessary regulation of this service ANACOM launched a national project aiming to implement a nationwide DVB-T monitoring system. A consortium formed by Ubiwhere and Wavecom (backed up by INESC-TEC) applied and ultimately won this application and thus was assigned for its development. This system is now in production for 4 years, and so the consortium decided to present their take on some of the design and implementation details.

The initial goal as briefly mentioned before was clear, consisting in measuring and performing analytics on the end users' (nationwide) perceived DVB-T signal in real time or as close as possible to real time. To achieve this, ANACOM defined roughly 400 deployment sites targeted for the instalment of the DVB-T monitoring probes allowing it to perceive the national coverage status of the digital television service. Regarding the division of tasks between Ubiwhere and Wavecom, Wavecom was responsible for the development of the DVB-T monitoring probe while Ubiwhere was in charge of the development of the central system (responsible for the gathering, persistence and processing of all the data of all the probes) as well as an installer application to be used by field technicians while installing/reinstalling the field probes. INESC-TEC's role was to carefully follow all the work ensuring the quality of the specified and ultimately developed solutions.

The current document focuses on the work performed by Ubiwhere and the components of which it was in charge.

On the side of Ubiwhere the team composition was formed by a pre-sales engineer, a project manager, a backend developer, a frontender, a mobile developer, a QA Engineer, a sysadmin and a UX/UI designer. Some of the base guidelines for the design of this system/deployment that had to be contemplated are presented below:

1. The system should be supplied as an appliance, no operation effort is to be required from the client
2. The data should be stored in-house. No data should be stored off the premises of ANACOM

3. The data should be transmitted using a secure channel as the Internet Service Provider (ISP) is potentially also the DVB-T operator and it must be ensured that there is no integrity tampering or snooping.
4. The dimension of the system should be such that it allows the storage of data from 400 probes, each one measuring 7 variables one time per second for a period of 2 years.

It is also worth to mention that, based on the results of this project the consortium (Ubiwhere and Wavecom), decided to create a new product named **rprobe (www.rprobe.com)** providing an evolution of this solution designed for spectrum sensing applications. It allows operators, broadcasters and regulators to remotely monitor the DVB signal in real time, keeping a record of the network status at any time.

2.Early design

The early design stages of the system focused mainly on the design and implementation of the field deployment tool, as well as the architecture of the backend.

FIELD DEPLOYMENT TOOL

The field deployment tool was developed as an Android application, and in a first stage available only on a custom image of Android OS on a tablet device. The fact that the developed application supported only a custom image of an android OS is related to some connectivity constraints that the DVB-T probe had in its configuration phase.

The end user of this application was the field technician. The field technician would start the deployment by answering a simple questionnaire, then would deploy the hardware on the field and connect the probe to the tablet.
Once connected to the probe the deployment tool would ensure that a data connection was correctly established by the probe, and would guide the technician in the final installation steps, such as adjusting the antenna and establishing communication with the central system over a secure VPN.

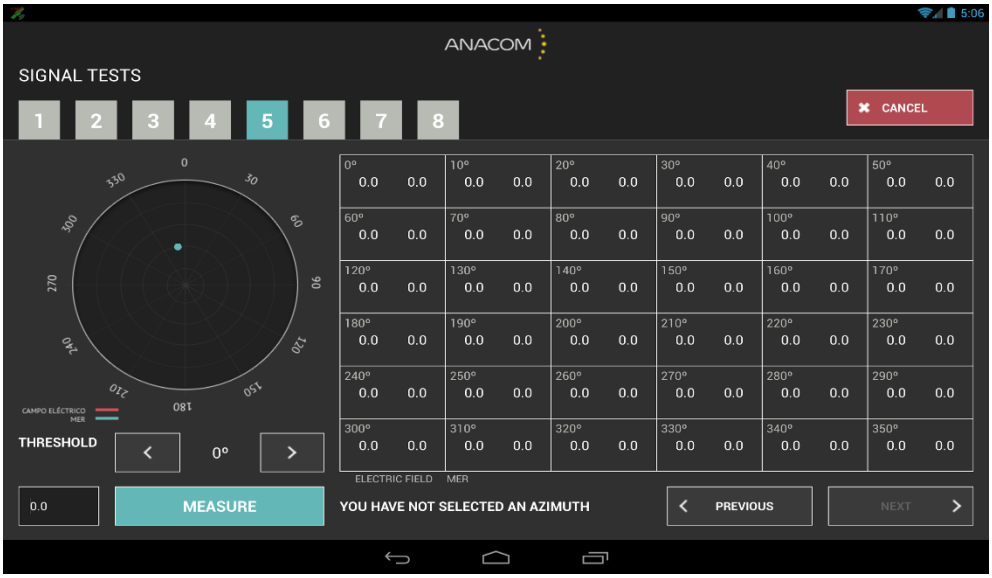

Figure 1: Azimuth configuration screen

The usage of this application was critical to ensure that the probe has a stable and secure connectivity channel with the central system as well that it is correctly configured ensuring that all the data that it collects has auditable quality. Critical steps such as the correct alignment of the probe's antenna are covered by this application and all this information is collected and sent to the central system allowing the creation of an install report for each probe, which can be accessed at any time to perform auditing actions.

3.Central System Infrastructure

The central system infrastructure is composed of seven 1U systems supplied by Asus. Two of those systems serve only as an IP frontend and VPN gateway. One is a NAS that is used to store a few non-critical shared filesystems and periodic on-site backups. The remaining four systems are all identical. Each has an instance of Gunicorm, Glassfish, Postgres, Pgpool and Cassandra.

All the systems networking is configured in a redundant way, so that if for some reason one has to disconnect a cable for a maintenance task, there would be no impact on the running system. Also, other than the NAS, every system is redundant - a failure or shutdown of a single node will not result in unavailability for the service.

4. Software Architecture

Other than the deployment tool, the system consists on a central API implemented in Java EE running on a Glassfish server, and a frontend server implemented in Django running on a Gunicorn.
The central API uses two databases, the first is a relational database that stores all the user information and the probe metadata, the second is a Cassandra database that stores probe measurements. Hazelcast is also used as a non-persistent data store for runtime locking mechanisms and job information.
At the time, we considered the possibility of basing our architecture in micro services, however some of the architecture patterns and tools that are nowadays stable were on their infancy at the time, so the choice was to go for a traditional glassfish cluster instead. The choice for glassfish instead of another application server was tied to the skillset of Ubiwhere's team at the time.

POLLING THE PROBES FOR DATA

The polling system periodically contacts each of the probes asking for a set of data for a specified period. The task should not be isochronous, it should start after a configurable amount of time after the previous instance of the task finishes. This pause between instances of the same task allows the system to run other tasks and avoid the possibility of overlapping requests to the same probe in case a previous request is still running after the interval.

On a non-distributed system this would be a trivial problem to solve. Just use one of many scheduling systems and create one recurring job for each probe. However, on a distributed system the problem becomes a lot more complicated. One option would be to use a gossip based solution to figure out which system is responsible for polling each probe. Unfortunately, java EE 7 does not have a standard way (that the team was aware) to get the instance id in a clustered environment, so we would also have to add a lot of complexity to achieve this. Another option would be to elect a master for the cluster, but that would introduce a lot of recovery issues.

The solution we elected was a bespoke system tailored to our needs. One that takes advantage of the characteristics of the system. For instance, if we always restart the whole cluster when performing maintenance why should we figure out a way to redistribute probes among nodes at runtime?

The system we developed periodically lists all the probes is the database in a random order, and tries to schedule a pooling task for each one of them. The scheduling task is performed at a slow rate on purpose allowing the other members of the cluster that are starting up at the same time to take for themselves some of the probes as well (taking advantage of a controlled race condition). When a probe is scheduled, a scheduling record is placed on the Hazelcast instance, hence when another node tries to schedule a given probe, it can skip it if a recent enough record exists. Given that the task runs periodically, it will automatically re-schedule the polling for any probe that is left orphaned as the scheduling record ages past a set amount of time. The polling task itself checks what is the latest measurement polled from the probe, and requests up to 6 hours of measurements from the probe internal storage (the probe can hold about two weeks of measurements). It also performs sanity checks on the retrieved data and stores it on the Cassandra database, and finishes with the re-scheduling of a polling task for that probe, overwriting the record on Hazelcast with a fresh one.

There is no hard-locking mechanism that avoids dual scheduling. If for some reason the probe gets scheduled on two nodes at the same time, it is likely that both will perform the scheduling task. However, it is highly unlikely that the tasks will finish at the same time, in part due to the design of the probe. This means that after the first scheduling collision, when trying to re-schedule a task, the last node to finish will read a very recent record from Hazelcast and skip the rescheduling step. The fact that the probe was asked twice for the same measurements is also not a problem as this is very uncommon, and the returned data will be the same, so the only problem would be the additional data consumed if this was a common situation. The same race condition is allowed during the initial bootstrap process, where each node on the cluster takes over a random set of probes.

STORING MEASUREMENT DATA

Measurement data is stored on a four node Cassandra cluster. This cluster was originally installed on Cassandra 2.0, and is now running Cassandra 3.0. It was always upgraded while running in production and throughout the whole life of the project, we only needed to wipe a node once due to an unspecified problem that would severely degrade the performance of that single node. The cluster is repaired every night during the off hours, and from time to time we perform a rolling restart without interruption to the service.
However, the operation of the Cassandra cluster was not always so smooth. Early versions of Cassandra would sometimes hang during the repair process, leading to a

less than controlled restart of the node or sometimes the cluster. While upgrading we also faced some issues upgrading our sstable binary format, a process that takes a very long time when each node contains more than one terabyte of data.
An early decision that we performed on this system was to use an ORM on top of Cassandra, specifically one that implemented the JPA specification. The original rationale for this decision was that this would allow a new developer to easily understand the data model and how to operate on it. On the more recent rprobe implementation, this ORM was abandoned in favour of a more stable driver.

POST PROCESSING OF THE COLLECTED DATA

In order to allow the client to visualize data over long periods of time, in an effort to detect long time trends, the received data is post processed after being stored. The post processing process is executed at 15 minute interval, and it gets information from all probes that have at least one complete hour of measurements. For each variable, the system calculates the median and average as well as standard deviation (linearizing the data when the data is logarithmic) and calculates a 100 bucket histogram of that data. This pre-calculation allows the client to visualize long term trends once they become significant. The results are also stored on the Cassandra datastore for later retrieval.

5.Operation

THE LIFECYCLE OF THE COLLECTED DATA

After being collected from the probes, the data is stored on a Cassandra datastore. This allows our client to visualise the data and replay it on a complex event processing system that enables advanced analysis on the collected data. However, disk space at private facilities is not infinite nor is it easy to upgrade. For the first year of the system operation, the development team kept a close eye on the evolution of the disk space usage, and started a plan to remove data while keeping the largest possible dataset on the system datastore for online analysis.

The plan is simple, let the data grow until it uses about 40% of the available disk space, then use those remaining 10% (avoid going over 50% as a scrub may require that much free space) to write tombstones that will eventually delete 10% of the

existing data when the sstables are compacted. However, the Cassandra datastore way of operating is sometimes unpredictable, with tombstones not being removed until the sstable that contains the data is compacted with the sstable that contains the tombstone.

A backup tool was designed when the version of Cassandra was still 2.2, and range deletes were not supported, hence a single delete statement must be created for each deleted record. As the system is updated to the latest 3.0 release, the development team may review this strategy after some tests.

The backup tool creates an export directory with the exported data and the respective delete statements. Those delete statements are moved to a second directory, and the exported data is compacted and moved to an external cold storage system. In order to allow the data to be easily handled should the need arise, the data is chunked into one month periods and compacted on disk.

A BRIEF HISTORY OF THE SYSTEM

The system design allows for the failure of every single component but the top of rack switch or the internet connection (those are the only single point of failure that we are aware of to this day). The initial system setup was based on Ubuntu server 12.04, Java 7, Cassandra 2.1 and Postgres 9.1. During its production run, all of these were updated at least once, in some cases with extensive work involved.

Updating cassandra

Early during the production run our dataset grew very quickly, and while trying to run some routine repairs on the cluster we came into pretty much every bug you can find on Datastax's issue tracker (not really but at times it felt like that)! During our first months into the production run, we also figured out that a choice elsewhere was preventing us from performing a rolling restart of the cluster without causing application downtime. At this point the operations team was pretty much spending a lot of time trying to figure out how to increase the system stability, applying every single update to the datastore and hoping that the show stopper bug (not always the same) that was keeping the cluster from operating at full potential would go away. After each update we attempted to run the repair, scrub or cleanup tasks, and always for one reason or another the task would fail. Here, we must praise datastax, as their bug tracker and community support was always stellar. Usually when we hit a snag, a

quick bug tracker search would quickly reveal the fix was already committed and due for release soon.
After the update to Cassandra 2.2, (and the migration to JDK 8 from oracle), the stability of the system began to increase, and for a long time, regular operation run scripted from cron without human intervention.

The update to Cassandra 3 was performed during a major system revamp that allowed us to migrate from Kundera's thrift driver to Datastax's native driver, which gave us a significant performance increase as well as the ability to perform rolling restarts of the cluster.

One near miss

While operating Cassandra 2.2 (an early release of 2.2), and holding about 1TB of data per node, the system became quite sluggish. After a series of tests, we determined that one of the nodes was systematically timing out queries. The first thing we tried was to repair the node partition range. The repair failed due to one of the bugs referenced earlier. Then we tried a node scrub, however scrubbing that amount of data takes quite a long time and eventually after about a week we gave up and assumed there was something wrong with the data. Amazingly, this node was not writing any error messages to the log files that would help us determine why it was misbehaving.
After long consideration, we decided the best course of action would be a full node rebuild. Doing so exposed our client to a situation where another node's failure would cause data loss as our replication factor is 2 and the cluster is composed of four nodes.
We shut down the node, removed its data directory and restarted it. After some struggle, the node joined the cluster, and a repair process rebuilt its sstables from the data stored on the remaining nodes. This whole process took about a week, we kept the rate slow to avoid impact to the running system. After repairing the whole cluster, we ran a scrub and cleanup on each node, ensuring that data that was added while the cluster had only three nodes was moved to the proper nodes and no garbage remained on each node.
After this episode, up to the time of writing this document, this was the single big issue we faced on the whole cluster that caused some concern both to us and to our client.

From Ubuntu 12.04 to 16.06

As ubuntu has a very short support cycle for an enterprise linux, we started planning the migration from 12.04 to 16.04 about one year before the EOL 0f 12.04. We performed plenty of tests, and tried to ensure we knew exactly what could go wrong and how could this process fail. We simulated the upgrade of every single software package we were running, from the parts of the service, to support software such as keepalived and pgpool. We were sure we could go ahead, and about one month before the EOL of 12.04, we scheduled a week of downtime and started working on the update.
If anyone ever tried an upgrade such as this on a long running traditional (not infrastructure as code) system, I think that person can understand the stress that the team feels when we've started updating the first node, and most of all the sensation of pure panic when the update fails! Of course it failed, it was the only thing that never happened on our test systems!
After a lot of debugging we figured out the cause of the failure. We needed to uninstall anything related to pgpool2, to java 7, and our monitoring system (Icinga) from each node before performing the update. The update was performed in two steps. The first step was to update from 12.04 to 14.04 and migrate the Postgres cluster from 9.1 to 9.3, the second step was updating to 16.04, migrating the Postgres cluster from 9.3 to 9.5, and writing a few systemd units to start our servicers on boot. Repeat the process six times (4 application nodes and 2 frontend systems) and we should be good to go right? Wrong! We removed our pgpool system, and needed to rebuild it. We took the opportunity to rebuild the pgpool with streaming replication instead of query replication by pgpool, we find this methodology a lot more reliable and since setting it up we never had any failure.

From Glassfish to Payara

When the service was first deployed it ran on a Glassfish 4.0 application server. With time, it was updated to 4.1, but the reality is that the Payara release is a lot more stable, and it packages a managed instance of Hazelcast, a service that we use to maintain state across the cluster where needed.
The update of the system to run on Payara required a bit of application development, we stopped launching our embedded Hazelcast instance, and started injecting the managed instance where needed. Overall, this decreased the deployment time of the war and reduced the applicational code a bit, something that always helps with maintainability.

6. Conclusions

This whole process was (and keeps being!) a learning experience for our team. From system design to architecture, to operational procedures or client expectations management. This system proved that a small team can build reliable software that has a huge impact on the quality of a public service on a country. We, at UBIWHERE are proud to be a part of it and of our contribution to increase the quality of the DVB-T signal in Portugal. By using this solution, ANACOM is now able to monitor 24/7 the quality of the digital television service from the end user's point of view. With this tool, the regulator can detect potential problems even before a complaint is reported by a citizen. Furthermore, for each complaint, ANACOM is able to consult this monitoring system and assess if this problem on the user's own tv installation (and if confirmed, help the user to correct its installation) or, in fact, a problem of coverage to be solved by the operator. In what regards the initial considerable amount of problems reported by users, ANACOM was able to collect a set of monitoring data and provide detailed reports to the service operator exposing that there were real problems in the digital television signal grid. In fact, the granularity time stamping, and contextualised features of the monitoring system allowed ANACOM to detect signal failures associated with weather conditions, tides as well as signal collisions between multiple signal transmitters. These problems would be ultimately impossible to detect without such a monitoring system since they are only perceived by the population and therefore sensed at certain (and sometimes short) time of the day. With this powerful tool, ANACOM already asked for a considerable set of interventions in the television digital broadcasting system by the service provider thus considerably increasing the quality of this service in Portugal.

As mentioned in the beginning of the document, meanwhile a new product was created based on the building blocks of this solution. **rProbe (www.rprobe.com)** is an evolution of the previously described system being able to monitor a wider spectrum of radio communications. Both the monitoring probe (rNode) as well as the central system (rCenter) benefited from the experience gathered in this project and were enhanced with extra functionality in order to be deployed in other ecosystems that do not use DVB-T. rCenter was also evolved to a multi-tenant solution and can now accommodate more than one client in a cloud environment. However, due to some constrains of potential clients, rProbe solution can still be deployed in a private client environment (such as what was done for ANACOM) without losing any of its functional and non-functional features.