Programme opérationnel Interreg IVA

France-Espagne-Andorre 2007 – 2013

SISPYR

Sistema de Información Sísmica del Pirineo

Système d'Information Sismique des Pyrénées

Sistema d'Informació Sísmica dels Pirineus

Near real time data exchange
Action 2.2

Main contributor: IGC	2013-04-24

Authors: Jose Antonio Jara (6), Javier Artero (6), Núria Romeu (6), Xavier Goula (1)

Keywords:

Real time, data exchange

---

1. IGC, Institut Geològic de Catalunya
2. OMP, Université Paul Sabatier, Observatoire Midi-Pyrénées
3. UPC, Universitat Politècnica de Catalunya
4. IGN, Instituto Geografico Nacional
5. BRGM
6. GEOCAT

Synopsis

The aim of this action is to provide a service which enables the temporary, sorted and centralized storage of velocity and accelerometric data in real time from the stations belonging to different data centres. It’s necessary to develop an API which allows the request of these data by means of an automatic system for further near real time processing (e.g.: in case of an event).

This system will grant data centres a shared data pool, and thus the results from processing the data will be similar without mattering which data centre operates with it.

Incidentally, this centralized design will free the bandwidth for inter-data centre communications.

Previous to this writing, research has been conducted to determine the availability of already developed solutions that would fit the current requirements. These solutions have been considered under these criteria: open source, free source, accessible, up-to-date and easily deployable and maintainable.

Contents

List of illustrations

List of tables

1. System requirements

1.1. *Used Terms*

To clarify and avoid misunderstanding some terms will be defined.

The terms “*data*”, “*waveform*”, “*waveform data*”, etc. are referring to seismic and accelerometric waveform data obtained from seismic and accelerometric stations involved in SISPyr project.

The terms “*repository*”, “*storage*”, “*archive*”, etc., all refer to a centralized facility where seismic and accelerometric waveform data obtained from seismic and accelerometric stations, along with station metadata, which comprises station coordinates, sample rate, height, etc., are stored under a certain conditions determined by the requirements exposed in this document.

The term “*data centre*” refers to all the SISPyr Seismic Data Reception Centres: IGN, BRGM, OMP, and IGC.

The term “*module”* is used discretionally when referring to module or sub-module, for the sake of simplicity.

The terms “*feed link*” and “*data feed*” are used to describe a single network connection transmitting a constant stream of waveform data associated to one or more channels.

The term “*wildcard*” refers to characters in a string that can be substituted for any other character or characters. In this module only two wildcards will be used. The question mark (“?”) can be substituted for any other character, while the asterisk (“*”) can be substituted for zero or more characters.

1.2. *Measuring units and formatting*

This section defines which units will be used when dealing with measurable terms that appear through the document.

Unit for time measurement will be seconds with a decimal resolution up to milliseconds unless otherwise specified. Items such as latency or time intervals fall into this category.

Dates will be displayed in *dd/mm/yyyy* format, and stored in seconds passed since midnight from 01/01/1970, with decimal a resolution up to milliseconds

Any geographical coordinate reference will be dealt in degrees and minutes for latitude and longitude, either in display or storage.

Distances will be measured in kilometres, with decimal resolution up to meters.

Channels will be fully identified by a SCNL code in N.S.C.L format, where N stands for network code using up to 8 characters, S for station using up to 6 characters, C for component using up to 8 characters, and L for location using up to 2 characters.

In this format, an undefined location will take “--" value.

Additional support for SCN code will be implemented to preserve backward compatibility, using the same format specification as in SCNL.

1.3. *General description*

In order to grasp a better understanding of the module M2.2, it’s necessary to break down the whole into smaller self-contained sub-modules. And to define the boundaries of these sub-modules a closer look must be taken at the data flow and the motivation to develop this module.

A sample of current data flow for processing can be schematized like this:

- *Data centre receives data from RT stations.*

- *Data centre applies some processing to the received data.*

The desired data flow for processing would be represented by these steps:

- *Data centre receives data from RT stations.*

- *Data centre receives data from other suitable stations thanks to SISPyr module M2.2.*

- *Data centre applies some processing to the received data.*

Data centres gather seismic and accelerometric waveform data from stations through a variety of communication links. Then these waveforms get stored in real time into their servers.

If one data centre wants process data in near real time, right now it can only deal with incoming data from RT stations.

It’s obvious that the more available data, the better precision will be attained at processing, and to fulfil this availability of data a centralized repository must be present, where waveforms from stations in the area of interest can be exchanged between data centres in a near real time environment without overloading communications between them.

Attention must be paid since an environment where data is shared implies two flows separated away by concept and time: one to obtain data (*Requester*), another to serve and share data (*Provider*).

Therefore, this architecture effectively splits the module M2.2 in two development blocks defined by the kind of data flow involved.

1.3.1. Requester Overview

Before data gets stored into a centralized repository, it has to be sent there. This necessity leads to the definition of first sub-module, which will provide a unidirectional real time link from data centres to the shared storage, and will be tagged as *Real Time Feed* sub-module.

Through this module data centres will send waveforms, received from their stations, in a constant and reliable stream that will get stored into a centralized repository, represented by another sub-module named *Temporary Repository*, which will take care of the storage and maintenance of any data received through *Real Time Feed* module.

1.3.2. Provider Overview

Requester will leave the storage facility filled with (near) real time data from stations contributed by data centres. *Provider*'s task is to serve stored data to a client on the basis of a set of parameters, such as time interval and station name filtering.

1.4. *Restrictions*

1.4.1. Data centre systems

Existing data centre's data acquisition and processing systems must be taken into account when developing the solution, as those will determine the kind of input data that module M2.2 will get.

Most data centres have a SeiscomP Seedlink server, Nanometrics NAQS server or a Güralp Scream! Server.

Knowing this, the module M2.2 can take advantage of the availability of those services interfacing to them in order to establish the required data feed.

These servers have known and tried data transfer protocols, while offering the permanent streaming that this module requires.

By using these existing systems, a seamless and non-intrusive integration of the system towards the data centres is attainable, since the data input aspect of the module is in fact already deployed and functional at each data centre.

1.4.2. Input data protocol

Above restriction constraints the protocol employed to transfer waveform data from data centres to the *Requester*.

Seedlink, NAQS and Scream will be the data transfer protocols initially supported, but effort will be made to allow the addition of other protocols in the future.

The transport protocol will be TCP/IP, because of its reliability and error control.

1.4.3. OS and language

Chosen OS for module M2.2 development is Windows XP 32bits, and programming will be done under Visual C++.

1.4.4. Sub-modules location

Requester (Real Time Feed, *Temporary Repository)* and *Provider* will be running on the same machine, by the same user and under the same privileges.

Clients for the *Provider* will be a standalone sub-module working in local or remote environment.

1.5. *NRT server modules description*

1.5.1. Requester - Real Time Feed

The one and only purpose of this module is to deliver data feeds from the data centres involved in SISPyr project to the *Temporary Repository* module.

This single-purpose module should meet the following requirements:

- The module must provide means to establish feed links to data centres. Through these links, data centres will send the real time seismic and accelerometric waveforms that this module will redirect, once received, to *Temporary Repository* module.
    - There will be several protocols to deal with, so a variety of interfaces must be implemented to support each of them. These interfaces will communicate with the appropriate data centre's servers and establish data stream.
    - There should be a permanent stream of data. This involves that gaps derived from shortages in the streaming (not from the seismic data) must not occur. In case these become inevitable, there should be a mechanism to deal and ease the impact of gaps in the feeding and storage, implemented inside the boundaries of this module.
    - To assure the data stream reliability, a mechanism for checking the connection state must be provided in the form of keep-alive messages between data centres and the module. This will allow the detection of network failures and offer a chance to repair them if possible or notify by means of error logging.
    - An optimum bandwidth usage would be desirable. This implies the usage of a compression algorithm to reduce the size of data sent from data centres, and consequently a decompressing algorithm applied to the data before being transferred to the *Temporary Repository* module. Balance between bandwidth and processing time must be retained, so the compression should provide enough benefits in the bandwidth aspect to compensate the time/processing invested in compressing the data.

    - Rational usage of network resources is required. As data centres will be providing a service to the *Real Time Feed* module comprising a range of channels, it should be considered enforcing data multiplexing. That is, allowing multiple channels' data to travel through a single feed for each data centre.

      (Relation → 1 feed = n channels)

- A way to configure the behaviour of the module must be provided so it becomes a scalable solution.
    - Ability to create or remove feed links.
    - Ability to edit existing feed links' properties such as associated data centre and related channels.
    - Any setup change should be transparent from data centre's point of view.
    - Configuration will be done through text files. Those text files will contain all the configuration parameters in human readable format and will be stored in sub-module's local machine.
    - Modifying configuration will imply restarting this sub-module for changes to take effect.
    - The sub-module must be prepared to allow the addition of new data transfer protocols. This means that protocols can't be hardcoded inside the module and a plug-in interface must be provided for expandability and interoperability.
- Should be an errorless module. As this is unattainable, it must be done error-proof.
    - Errors and warnings must be logged in a log file in human readable format. Some configuration mechanism must be implemented to choose or modify log file path.
    - In case of a fatal or unrecoverable error that causes the module to crash, some kind of watchdog-type mechanism must be provided to revive the module. Data feeding module state will be constantly monitored by this watchdog-type mechanism. Once alive, the module will try to re-establish the ongoing feeds when crash took place. This process should be unattended and completely automatic.

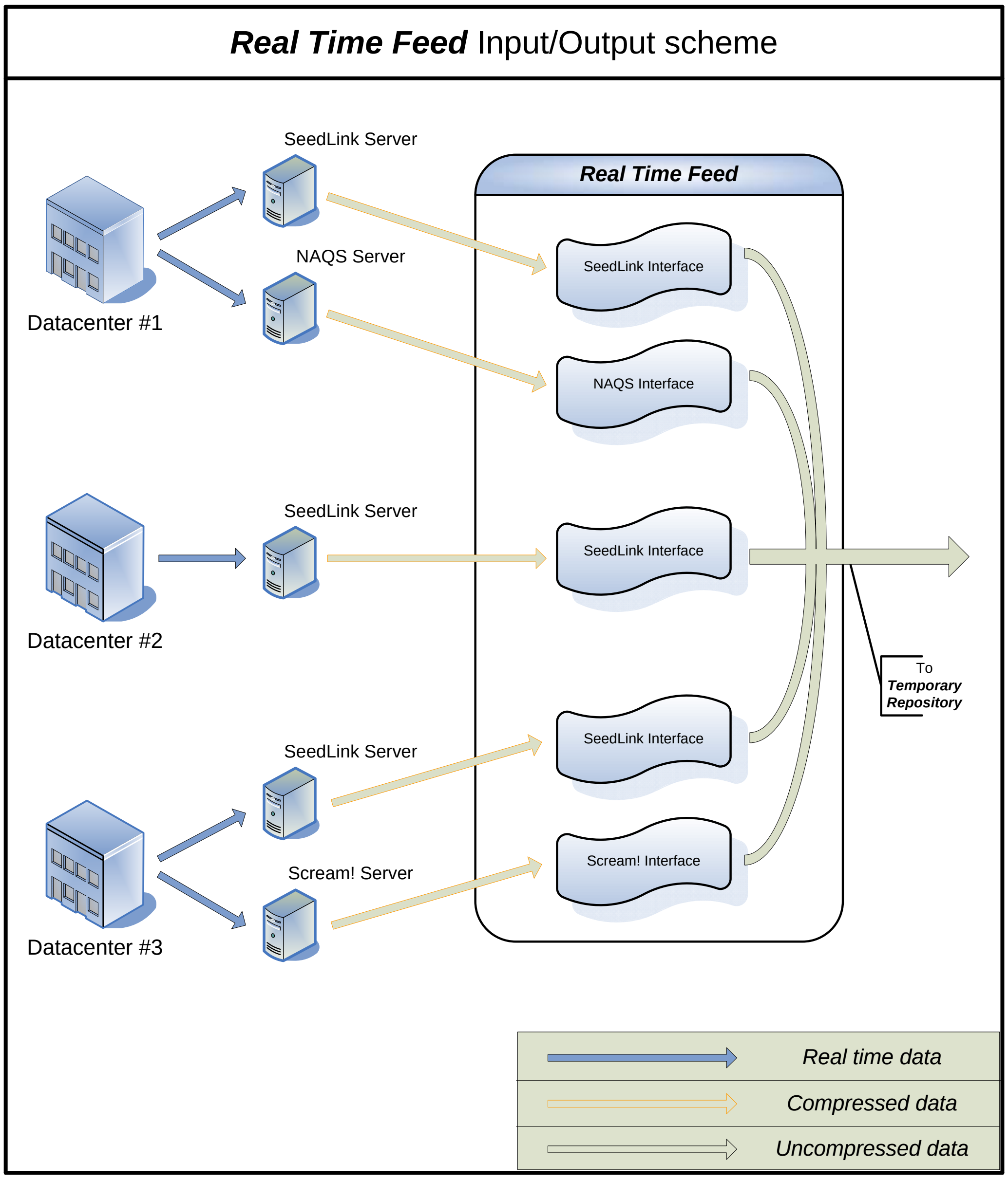


Figure 1 - Requester's real time scheme

1.5.2. Requester - Temporary Repository

All data received by *Real Time Feed* module must get saved somewhere. That's why a *Temporary Repository* module must be defined by means of a set of requirements:

- This sub-module must store the data from *Real Time Feed*, so it requires repository system.
    - It will use the chosen OS' file system. Being Windows XP 32bits, NTFS will be used rather than FAT32, to get rid of file size limitations.
    - One file will be assigned per channel. This will mean a more accessible searching and maintenance.

        - Files will be named after the associated channel in SCNL format.

        - The "near real time" nature of the project negates the need of storing the full time wide waveform, therefore only a section of gapless data will be included into the file.

          The length of this section will be configurable, with a default value corresponding to 1 day of gapless data. This data interval will be the same for all channels/files.

        - The file will behave like a circular buffer, where older data gets overwritten by newer incoming data. Some mechanism must be provided to manage these overwriting operations properly.

        - The files will be accessible directly by the *Provider*. Storing the data by means of NTFS file system implicitly allows this.

    - Waveforms received from *Real Time Feed* will arrive and be stored in uncompressed raw format, for the sake of access speed in spite of storage space availability.

- Must support the *Real Time Feed* feeding from different and simultaneous sources and, at the same time, serving to the *Provider* different and simultaneous data requests. This implies a strong control of concurrent access to the stored resources.

    - A list of channel properties (metadata), such as their geographical coordinates, will be maintained in the *Temporary Repository*. It will be stored in a database that allows easy updating and maintenance, and independent access from waveform data.

- Must be an error-proof module

    - Errors and warnings must be logged into a log file in human readable format. Some configuration mechanism must be implemented to choose or modify log file path. This path will be restricted to the local machine running the module.

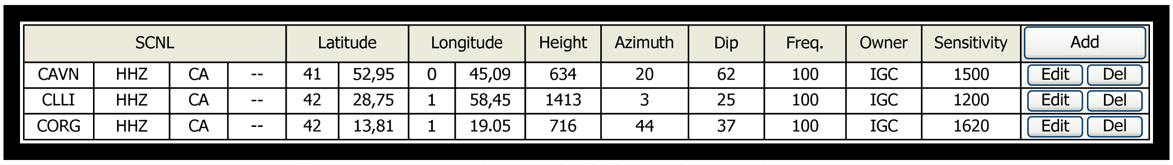
    - In case of a fatal or unrecoverable error that causes the module to crash, some kind of watchdog-type mechanism must be provided to revive the module. Once alive, it will try to re-establish the communication with *Real Time Feed* module. This process should be unattended and completely automatic.

    - Any storage system, upon suffering an error, might be prone to data corruption. Recovering from corruption is a bothersome task, so some kind of prevention must be implemented in the form of redundancy in file operations and file integrity checking. It will allow an easier recovering if any file gets corrupted, since an up-to-date copy will be present.

- A way to configure the behaviour of the module must be provided so it becomes a scalable solution.

- Ability to create or remove files linked to specific channels. *(Relation → 1 file = 1 channel)*
- Ability to change the time interval limitation for the files. Doing this will lead to the creation of a new set of files, destroying any old files and consequently their contained data.
- Ability to choose the behaviour in case of an error that can't be automatically repaired: Stopping the module or ignore and continue.
- Any change in the properties from an incoming channel data (e.g.: sample rate) will imply the creation of a new file for that channel, destroying the old channel file and its data. This will assure consistency on stored data.
- The implementation of a file corruption preventing system (as mentioned in the "error-proof" section above) negates the ability to change already created file paths freely. Doing that, will imply the creation of new files.
- Configuration will be done through text files. Those text files will contain all the configuration parameters in human readable format and will be stored in sub-module's local machine.
- Modifying configuration will imply restarting this sub-module for changes to take effect. Additionally, depending on the modified parameters it might imply recreating the files affected by the changes and losing the old data.

- An admin tool should be implemented to manage station's metadata database. This admin tool will be based on a web interface and tested over IE7, IE8, Firefox 3 and Mozilla.
    - Will allow adding, editing and deleting channels and its properties, by means of a web form. The following properties will be stored:
        - SCNL code
        - Latitude & longitude
        - Elevation (measured in meters)
        - Dip and azimuth (measured in degrees)
        - Sample rate (measured in Hz)
        - Owner
        - Sensitivity in 1 Hz (counts / measuring units)
    - Preliminary layout:



SCNL				Latitude		Longitude		Height	Azimuth	Dip	Freq.	Owner	Sensitivity	Add	
CAVN	HHZ	CA	--	41	52,95	0	45,09	634	20	62	100	IGC	1500	Edit	Del
CLLI	HHZ	CA	--	42	28,75	1	58,45	1413	3	25	100	IGC	1200	Edit	Del
CORG	HHZ	CA	--	42	13,81	1	19.05	716	44	37	100	IGC	1620	Edit	Del

Figure 2 - Metadata administration web tool GUI

- o Access to the tool will be password protected. Once logged in, password will be editable.

- o Administrator will be the responsible of maintaining synchronization between channels in the list and active feeding channels.

- A monitoring tool will be developed to check active feeds state. This monitoring tool will be based on a web interface and tested over IE7, IE8, Firefox 3 and Mozilla.

    - o In order to allow data health monitoring a database will store incoming packets' latency along with any other properties that are suitable for the aforementioned monitoring. This database will be synchronized automatically with the stored waveform data so it only retains latency for existing data.

    - o The tool must show real time information in a line for each channel configured in the feed links.

        - ▪ Channel code (SCNL) → String(s) showing SCNL code of the channel.

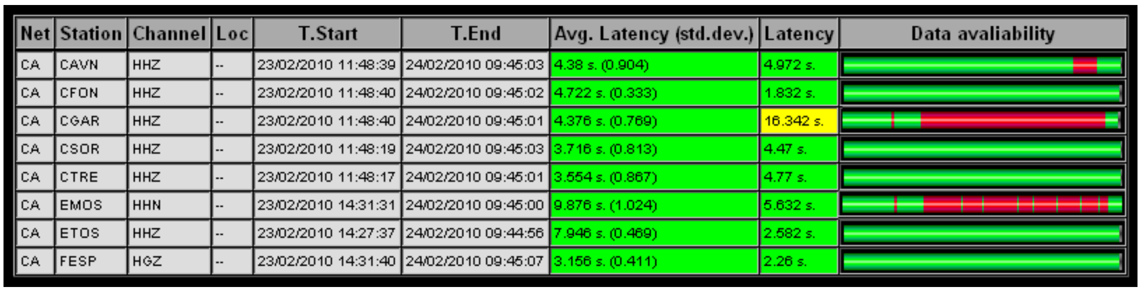
        - ▪ Latency → Display of average latency, average latency standard deviation, and latency from the last received packet, showing the seconds with decimal resolution of milliseconds, and a color code depending on the value: Green for values under 10s, Yellow between 10s and 60s, Orange between 60s and 300s, and Red over 300s. Latency is defined as the difference of incoming packets' timestamp and system local time. This time ranges are default values that will be configurable.

        - ▪ Data interval → Two fields, showing date of first and last available packets in *dd/mm/yyyy* format.

        - ▪ Graphical feedback about stored data for this channel in *Temporary Repository*, showing a bar with green areas for available data and red areas corresponding to gaps.

        - ▪ The list will be ordered by stations alphabetically and descending.

        - ▪ Preliminary layout:



Net	Station	Channel	Loc	T.Start	T.End	Avg. Latency (std.dev.)	Latency	Data avaliability
CA	CAVN	HHZ	--	23/02/2010 11:48:39	24/02/2010 09:45:03	4.38 s. (0.904)	4.972 s.	
CA	CFON	HHZ	--	23/02/2010 11:48:40	24/02/2010 09:45:02	4.722 s. (0.333)	1.832 s.	
CA	CGAR	HHZ	--	23/02/2010 11:48:40	24/02/2010 09:45:01	4.376 s. (0.769)	16.342 s.	
CA	CSOR	HHZ	--	23/02/2010 11:48:19	24/02/2010 09:45:03	3.716 s. (0.813)	4.47 s.	
CA	CTRE	HHZ	--	23/02/2010 11:48:17	24/02/2010 09:45:01	3.554 s. (0.867)	4.77 s.	
CA	EMOS	HHN	--	23/02/2010 14:31:31	24/02/2010 09:45:00	9.876 s. (1.024)	5.632 s.	
CA	ETOS	HHZ	--	23/02/2010 14:27:37	24/02/2010 09:44:56	7.946 s. (0.469)	2.582 s.	
CA	FESP	HGZ	--	23/02/2010 14:31:40	24/02/2010 09:45:07	3.156 s. (0.411)	2.26 s.	

Figure 3 - Feed data monitoring web tool GUI

- Interaction with this monitoring tool will be reduced to read-only operation. It won't be used to alter in any way the feeds or their configuration.

- The webpage will refresh automatically the information every X seconds, being X a configurable value.

- To provide some flexibility a server-side configuration will be available covering the following parameters:

  - Ranges of latency. There will be four ranges to configure, corresponding to green, yellow, orange and red display colours. Values will be in integer seconds.

  - Page refresh timer. A value in seconds indicating when the page will refresh with new information.

  - Channel filtering. Will allow specifying a list of stations, with the optional use of wildcards, which are allowed to be displayed. Channel codes will be SCNL.

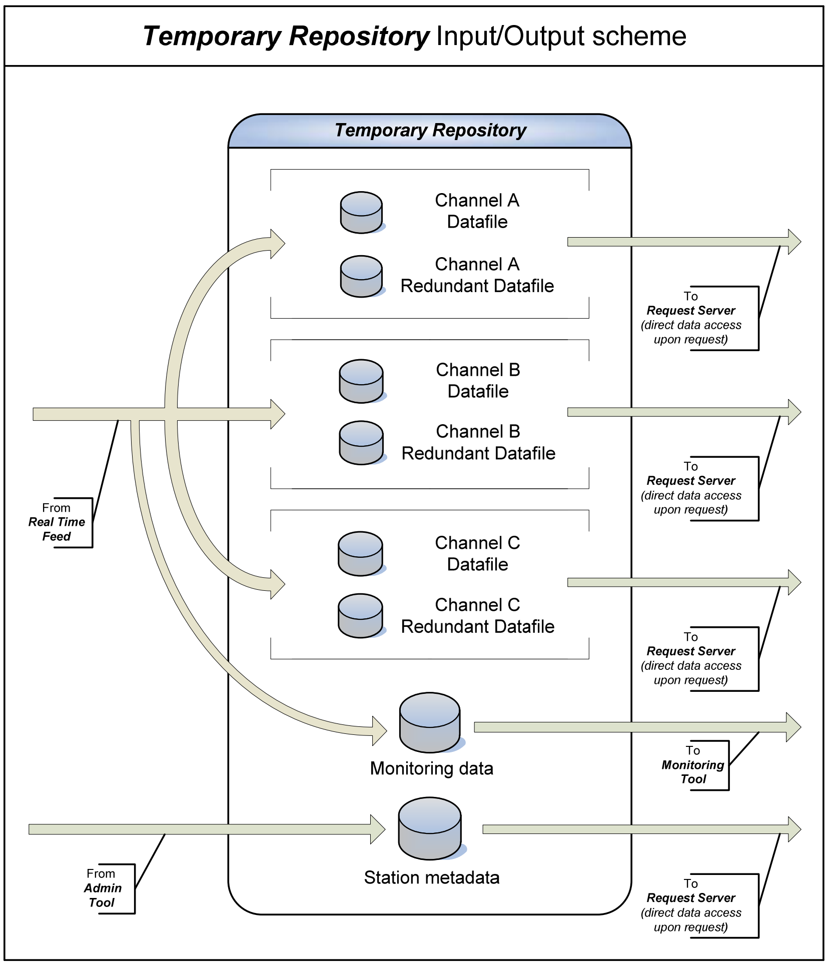

Temporary Repository Input/Output scheme
Temporary Repository
Channel A Datafile
Channel A Redundant Datafile
Channel B Datafile
Channel B Redundant Datafile
Channel C Datafile
Channel C Redundant Datafile
Monitoring data
Station metadata
From Real Time Feed
From Admin Tool
To Request Server (direct data access upon request)
To Request Server (direct data access upon request)
To Request Server (direct data access upon request)
To Monitoring Tool
To Request Server (direct data access upon request)

Figure 4 - Temporary repository scheme

1.5.3. Provider

Remote access by clients to stored data by *Temporary Repository* module will be granted through *Provider* module.

The following requirements define the *Provider* server role:

- Being a server, it must be able to provide a number of services to clients.
  - Main service will consist in attending a request from a client, formed by a set of parameters (channel code in SCNL format and a time interval) and returning to the client the data from the *Temporary Repository* that corresponds to the parameters.
  - Additional services will be implemented:
    - A service to provide a list with available channels stored in *Temporary Repository*.
    - A service to provide an expanded list with available channels and time intervals from the *Temporary Repository*.

        - A service to provide the list of channels with associated metadata.

    - Services will be synchronous: A service will receive a request, issue a reply and then process next request.

    - After establishing the connection with a client, the *Provider* will spawn the appropriate service to deal with requests sent by the client.

    - To support simultaneous client requests, any number of *Provider* service threads will be launched to satisfy the demand. This will be automated and completely transparent from a client-side point of view.

      (Relation → 1 Provider = n Service threads = n Clients)

- Upon request, has to obtain low latency access to the data

    - Validation of parameters defining the requests will be a server-side task. This means that the clients might implement a validation mechanism, but server will be the final responsible of asserting them, informing the client if there's any incompatibility towards the data stored.

    - This module must cooperate with the *Temporary Repository* module to control the concurrent access to stored data. Access to the requested data will be granted based on a first-to-arrive first-to-be-served policy, in order to accomplish the concurrent access needs. This will happen anytime there are overlapping requests on the same data.

    - Since data will be stored in files named after the SCNL format, finding the requested channel will be a trivial task. The main delaying factor will be isolating the data section that fit the specified time interval, so some kind of indexing system will have to be implemented to ease the lookup process.

- Once data has been retrieved, it must be sent to client.

    - Continuous streaming to the remote clients won't be allowed since it defeats the purpose of this module.

    - To achieve the desired fast servicing on the bandwidth aspect, a layer of compression will be applied. So requested data will be retrieved in raw uncompressed format from *Temporary Repository,* compressed, and then sent to the client.

    - Data will be sent sequentially, each sequence being one full interval of channel data. So if a client requests data intervals from 4 channels, it will receive 4 "blocks" of data sequentially. Therefore, no multiplexing will be applied to the transfer.

- A way to configure the behaviour of the module must be provided.

- Ability to configure the network parameters that define a server, such as listening port and timeout values for potentially dead connections.
- Ability to point the location of the *Temporary Repository* data so direct access can be accomplished from this sub-module.

- Must be an error-proof module.
  - Errors and warnings must be logged in a log file in human readable format. Some configuration mechanism must be implemented to choose or modify log file path.
  - In case of a fatal or unrecoverable error that causes the module to crash, some kind of watchdog-type mechanism must be provided to revive the module. This process should be unattended and completely automatic. If possible, once the module is up and running again, a notification will be sent to clients connected at the time of the shutdown so they can resume data transferring.
  - A mechanism to inform from server errors to a connected client must be implemented. Any request that cannot be accomplished will return and error message to the client so it can notify the user.
  - A mechanism to prevent overloading the server by connections or excessive processing must be implemented.

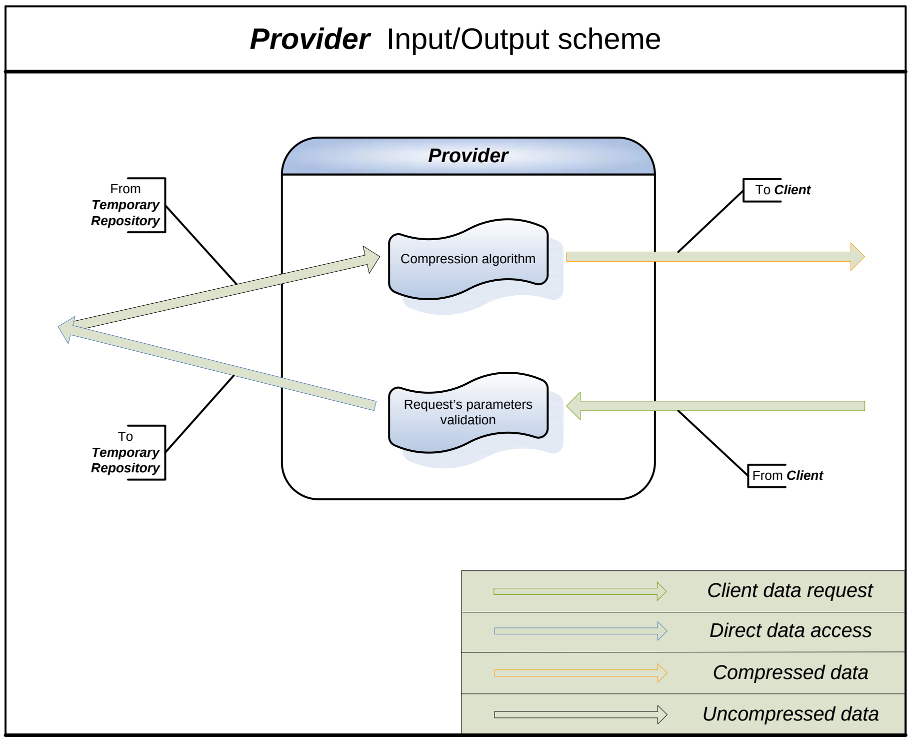


Figure 5 - Provider server role scheme

1.6. *Client*

An API will be provided to aid in the development of custom clients to fulfill the diverse necessities that might arise from data centres.

Nevertheless, a sample client will be created as an example of API usage and possibilities.

1.6.1. The Client API

This API must meet the following requirements:

- Must be usable in 32bits versions of Windows and Linux.
- Should encapsulate all the required functionality to interact with the *Provider*
    - Basic client-server functions:
        - A method to connect to a remote *Provider*, specifying server's IP address and port. This will establish the connection and leave it open, ready to send requests.
        - A method to drop the connection in a graceful manner, so the server can free allocated resources.
    - Methods for sending requests to the server:
        - A method that will allow requesting the list of currently available channels along their time intervals from the *Temporary Repository*.
        - A method that will allow requesting data from a particular channel and time interval.
            - Prior to requesting data, it will be necessary to ask for the channel list if not done before. Even if the list was already requested in a previous operation, it will be advised to request the list before sending any new data request due to the data constant update.
            - Parameters for data requesting will be the channel code in format SCNL, a time interval and a timeout for the operation.
            - Server will check channel and time interval availability, but client will also implement a preliminary validation. This will be attained doing a lookup trying to match request's parameters versus the obtained list of current channels and time intervals before sending a data request.
        - A method that will allow requesting the list of stations along with their metadata.

  - o Must be prepared to receive responses to the sent requests.
    - ▪ All requesting methods will be synchronous. This means that after sending a request, the method will wait till it gets the requested object (a list or a chunk of data) or a configurable timeout expires.
    - ▪ Requested data might arrive in compressed format, so a decompression shall be applied before any further processing is done. Once uncompressed, waveform will be in raw format.

- All requesting methods must return the result of their execution.
  - o Will be either and OK response, an error or in some cases a warning.
  - o It will be up to the client implementation to show a feedback to the user and deal with the error or warning.
  - o Internally, requesting methods might get an error response from the server. This should be delivered in a separated way from the client-side errors, since both error types could occur in a single method call.
- A set of utility methods will be also implemented:
  - o A wildcard channel filtering method. Will accept an input string with filtering wildcards and return a list of channels that match the filter.
  - o A coordinate-distance channel filtering method. Will accept an input geographical location and distance and return a list of channels inside the area defined by the circumference with centre at given coordinates and radius equal to the distance.
  - o A rectangular area channel filtering method. Will accept two input geographical locations and return a list of channels inside the rectangle defined by the two coordinates.
  - o An exporting method. Will convert and save the raw data received from the *Temporary Repository* to mini-seed format.
- Full documentation for the API will be delivered, along with code snippets showing how to properly use the methods.

1.6.2. Sample client

The sample client will show how to make use of the API. This implementation will accept an event location (latitude and longitude in degrees plus depth in meters) and event origin time (year, month, day, hour, minute and seconds) as a command line parameter, so the client application can request relevant channel trace data related to that event, which will be gathered and dumped to mini-seeds.

Client workflow can be described as follows:

- Once an event has been given through command line, the client will ask the NRT server for a list of available channels in the Temporary Repository.

- A configurable channel filter will be applied to the received list leaving only those channels that match the filtering.

- Having the filtered channel list, event location coordinates, and a configurable distance, client will call the utility method that finds stations that lay inside the circumference defined by geographical coordinates and distance.

- Having the stations, a time interval is required before asking for the data. Two choices will be given to calculate this interval:

    - Event-wise interval, to obtain full range set of data:

        - Base time will be the origin time from the event.

        - Pre-event and post-event times will be configurable values that will be subtracted and added respectively to the event's origin time to obtain a time interval.

        - Calculated interval will be used in the data request for all channels.

    - Station-wise interval, to obtain only relevant set of data:

        - Based on a basic crustal model, and having event origin time and geographical coordinates, along with stations coordinates, wave arrival times to each station will be calculated.

        - Pre-event and post-event times will be configurable values that will be subtracted and added respectively to each station wave arrival time to obtain an individual interval.

        - Calculated interval will be used in the data request for each channel.

- Then the client will ask the NRT server for the data of the channels returned by the filtering method with the time intervals calculated in the previous point.

- Received data, in raw format, will be converted to mini-seed format and dumped to several mini-seed files, one for each channel.

    - A log file will be updated with information about processed events and mini-seeds created. Any error found during the process will be logged too. Additionally, the logging will be shown in the application console.

    - Configuration will be done through text file. These are the main client parameters:

        - Distance (radius in km.) that will be used when calculating stations inside a defined circular area.

        - Pre-event and post-event time modifiers to be used in calculation of channel's data time intervals.

        - Mini-seed naming convention and storage path. Will be restricted to local machine.

- o NRT server address and connection parameters.
- o Log file path.
- o Channel filtering using wildcards that will restrict channels used by this client.

2. System design

2.1. *General view*

The NRT Server has been split into two main development blocks:

- *Provider* module
- *Requester* module

The simplest view of this structure is an input-output sausage, as pictured in Figure 6. Actually it's a streaming-input/output-on-request, but this topic will be covered in the in-depth descriptions for each module. For now, let's keep the input-output sausage simplification.

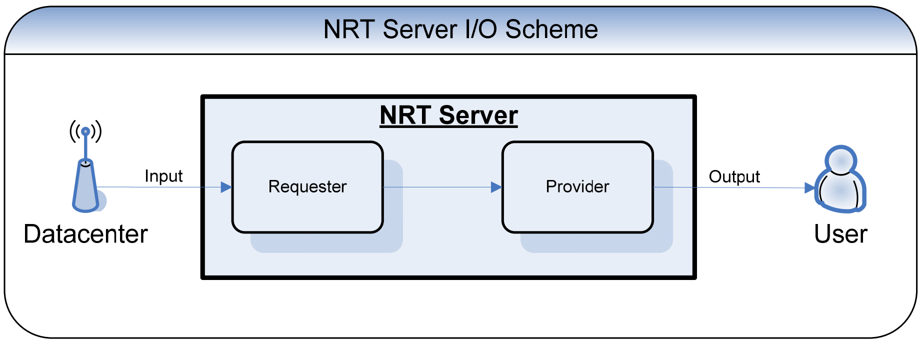


Figure 6 - The simplified input/output sausage

A review of existing seismic-oriented input/output systems led to the conclusion of taking advantage of the Earthworm model. Being an open-source solution it allows full customization and expandability to fit the needs of NRT Server, and provides a base environment that mirrors most of the requirements.

For the sake of features availability, Earthworm's version 7 onwards will be used, and through the course of this document Earthworm's modules will be added to conform to the full NRT Server requirements.

Below, the Figure 7 offers a panoramic view of the assembled module, its interactions with the data centres and users, a preview of the inner modules, along with the boundaries of the different sub-modules domains. Keep in mind that the user can be placed everywhere, so a client could be placed in a data centre or inside the *Provider* area.

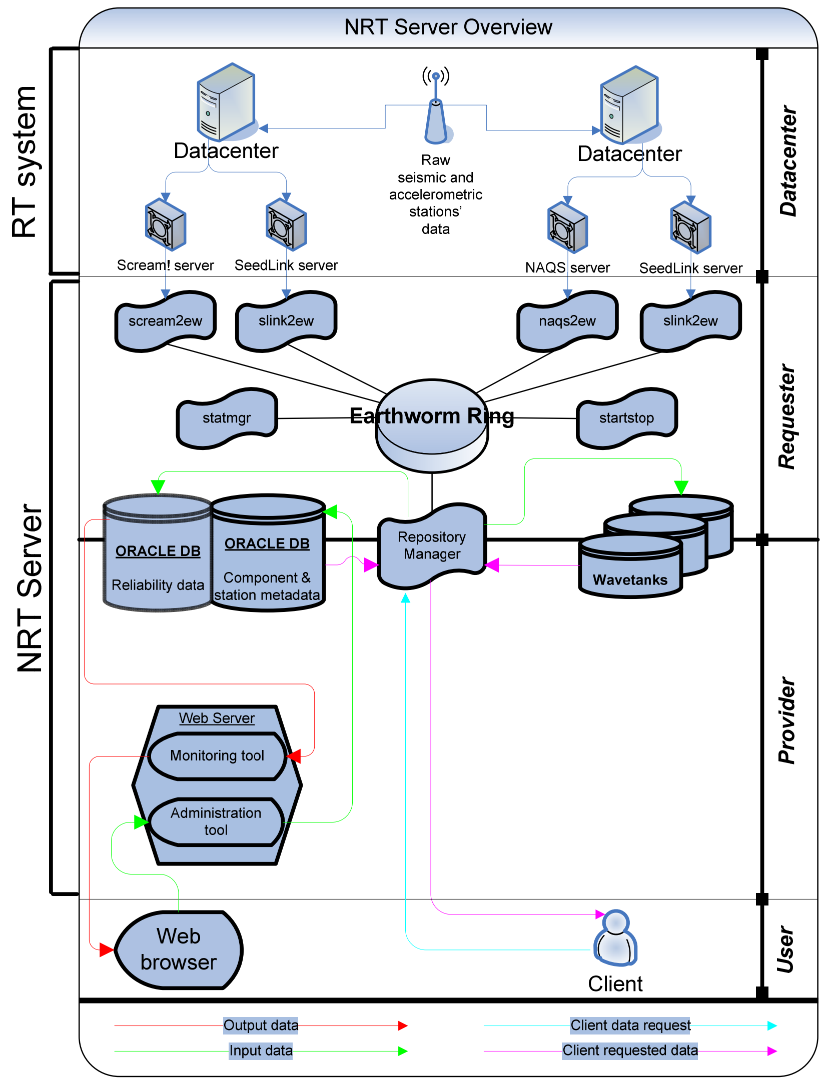


Figure 7 - NRT Server general view

2.2. *Requester*

Obtaining and storing data is the main purpose of this development block. Data needs to travel from selected data centres to be gathered in a centralized repository, so two tasks must be accomplished.

2.2.1. Real Time Feed

Each data centre has raw seismic data stored that must be outputted to NRT Server. Most data centres have streaming trace data output systems implemented, such as Seedlink, NAQS or Scream! Servers, so the NRT Server just must take care of establishing a link to these servers in order to receive the stream of data.

Knowing that Earthworm is the base for the NRT Server, some of its available modules can be used in order to get the data input in the system. Earthworm modularity allows expanding the data input diversity, but only those trace data output systems enumerated above will be initially supported.

- Seedlink support

  To establish the link to the Seedlink (v2.6) servers the module slink2ew (v2.0) will be used.

- NAQS support

  To establish the link to the NAQS servers the module naqs2ew (v1.5) will be used.

- Scream! support

  To establish the link to the Scream! (v4.4) servers the module scream2ew (v1.2) will be used.

For each available server an instance of the proper module must be created and linked. Once the needed modules are up and running, a stream of data will begin to flow to the system and the *Real Time Feed* will be enabled. An example of this situation can be seen at Figure 8 below.

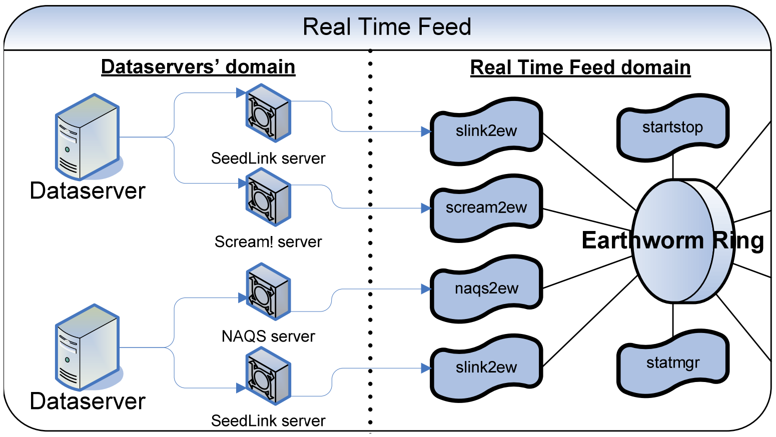


Figure 8 - An example of Real Time Feed setup

Data centres will transfer data to their Seedlink, NAQS or Scream! Servers, which will feed the slink2ew, naqs2ew and scream2ew modules, which will dump received data to the assigned Earthworm ring. Trace data traversing the Earthworm ring will do so in Earthworm's TRACEBUF2 packet format.

As long as this data feeds aren't stored they will be lost, and here's where the *Temporary Repository* is needed.

2.2.2. Temporary Repository

Once the trace data has been dumped into the Earthworm ring, it must be stored before it's lost. Earthworm offers a module, wave_serverV, designed to save trace data in files (called wave tanks) separated by the source channel, and serve them through TCP/IP links.

Besides trace data, there's other data that have to be saved: Data reliability and station/component metadata.
Natively, the wave_serverV module only supports storing and serving of trace data, so a customized version will be developed to fit our needs: the *Repository Manager*, and in order to store metadata and reliability data, a relational Oracle 9.2g database will be set up with full access from *Repository Manager*, by using the OCCI database API for read and write operations.

- **Station and component metadata**

Having station and component metadata available will be needed for data interpretation and processing, and *Repository Manager* will ignore any feed that hasn't associated metadata and will discard incoming trace data from it.
Metadata, though, isn't bundled in the packets received from the data centres, so it will have to be entered manually from the *Administration tool* (section 2.5**)** and will be stored in different Oracle database tables for station and component, namely the *Stations table* and *Components table*. This effectively gives the *Administration tool* control over which feeds are available to the users just by adding or removing the metadata register for established feeding channels.

To enable a feed, two kinds of metadata are needed:

- Station metadata, which encompasses those fields that describe the station.

- Component metadata, which defines individuality of the components from a station.

Data definition:

	FIELD	TYPE
Station	Station	String
	Network	String
	Latitude	Real
	Longitude	Real
	Elevation	Real
	Owner	String
Component	Component	String
	Location	String
	Dip	Real
	Azimuth	Real
	Sample rate	Real
	Sensitivity at 1Hz	Real

Table 1 - Metadata fields and data types defined for each station

Data insertion:

- An authorized user will create a register for a new station and fill up its metadata using the *Administration tool*.

- Once the station metadata is inserted, the user will create as many components (and their metadata) as needed. If a station has no components it will be ignored when receiving data from it.

- The *Administration tool* will take care of any database operations needed to register the data.

<u>Data edition/deletion:</u>

- The *Administration tool* will allow editing or deleting existing station and component records.

- Being stored in a relational database, all components owned by a station should be deleted before removing the owning station.

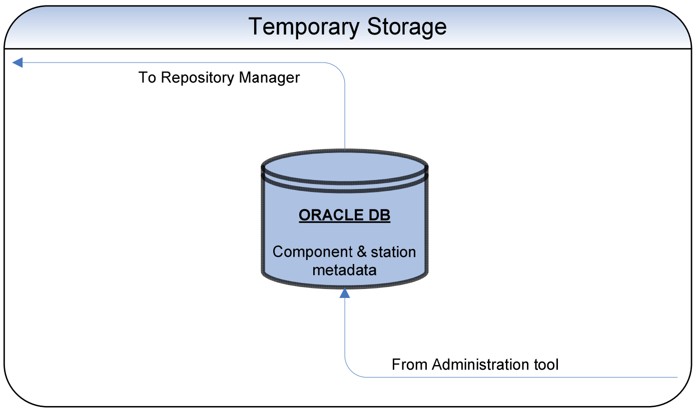


Image 1: Close-up of metadata storing

- **Data reliability**

With a remote streaming setup like the one needed in NRT Server, some means to check the state of these feed links must be implemented. A set of data that represents the reliability of the feeds will be continuously updated and will be reviewable by means of a *Monitoring tool* (section 2.6).

The healthiness of a feed link can be measured in three ways:

- Current latency, defined as the time elapsed between last received packet's data ending timestamp and entrance time of the packet into the *Repository Manager*. Storing this latency in real time from several channels simultaneously is too stressful for the database, so an optimistic approach will be applied, updating this value periodically.

- Average latency, being the average of all stored packet's latencies. Having an average value implies storing the variance and any other values needed for computing an average accumulation.

- Gaps in the received data. Keep in mind that trace gaps can be caused by a faulty connection between stations and data centres or between data centres and NRT Server. An excessive I/O stress on the *Repository Manager* can also create gaps.

This information must be synchronized to the available trace data. That means that the saved average latency and gap control will correspond only to trace data currently saved in the disk, and as soon as the information regarding a channel becomes obsolete it will be discarded.

Data definition:

FIELD	TYPE
Current latency	Real
Average latency	Real
Latency standard deviation	Real
Trace starting time	Real
Trace ending time	Real

Table 2 - Fields of monitoring system

Data insertion:

The *Repository Manager* will take care of inserting and updating the data reliability information in an automated process.

Data edition/deletion:

Data maintenance will be automated. The *Repository Manager* will monitor the data, and periodically will erase or update the existing registers that are out of synch with the data stored in the circular filebuffers.

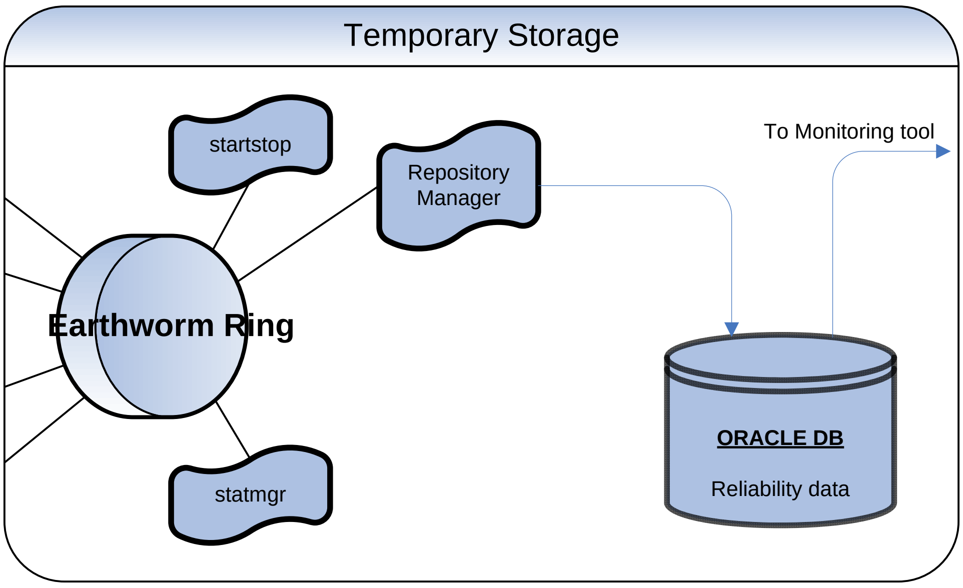


Figure 9 - Close-up of reliability data storing

- **Trace Data**

This is essentially the data received through the Earthworm modules that capture the streaming from the Data centres.

The *Repository Manager* stores that data in one file per channel, that behave as circular buffers, overwriting data older than a configurable “age”, by default 24 hours. Keep in mind that this “age” is measured discarding the time occupied by gaps between data.

Data insertion:

When a packet from an established feed gets into the Earthworm ring, the *Repository Manager* catches and inserts it into the corresponding file.
As pointed in *2.1.2.1*, only packets from existing channels in the Component & station metadata tables will be inserted.

Data edition/deletion:

Trace data can’t be edited or selectively deleted. Old data will get overwritten by the *Repository Manager* automatically when old enough.

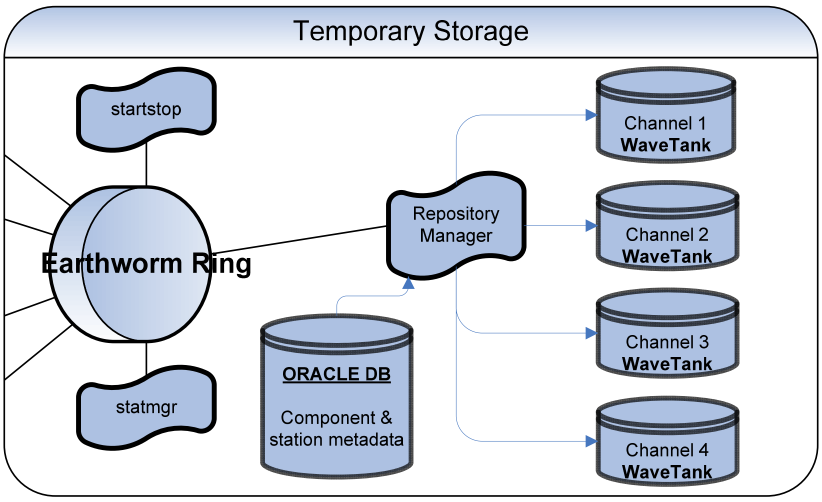


Figure 10 - Close-up of trace data storing

2.3. *Provider*

Having data in a storage facility implies offering means for accessing it. This is the *Provider*’s task.

The *Repository Manager* (described in section 2.2.2) will also be part of the *Provider,* since it will include a protocol to transfer trace data over a TCP/IP socket with a synchronous (blocking) behaviour. For every client that establishes a connection, the *Provider* will ask *Repository Manager* to spawn a serving process so it is able to respond simultaneous data requests from different clients.

The following services upon request are implemented:

- Returning a list of available channels, the time intervals of their data and their metadata. The list will include only channels with metadata in the *Stations table* and *Components table*.

- Returning a set of data restricted to a requested time interval for one or more channels, and streamed from the circular file buffers assigned to the requested channels.

Latency data doesn't need to be transferred since it's only used it to provide monitoring information, and will be accessed directly by the *Monitoring tool*.

To attain the maximum performance in terms of bandwidth, the *Provider* might add a layer of compression previous to sending the data via TCP/IP.

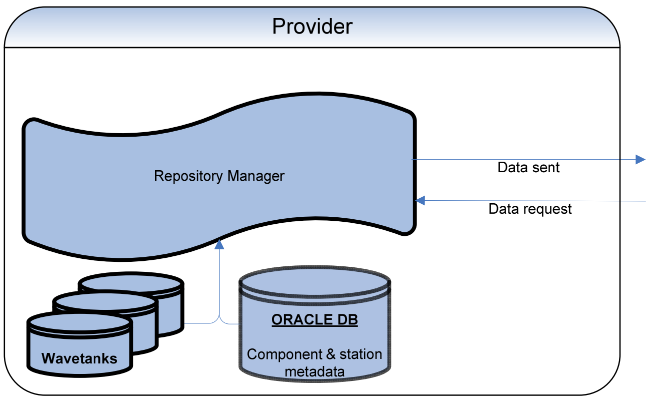


Figure 11 - Provider data flow

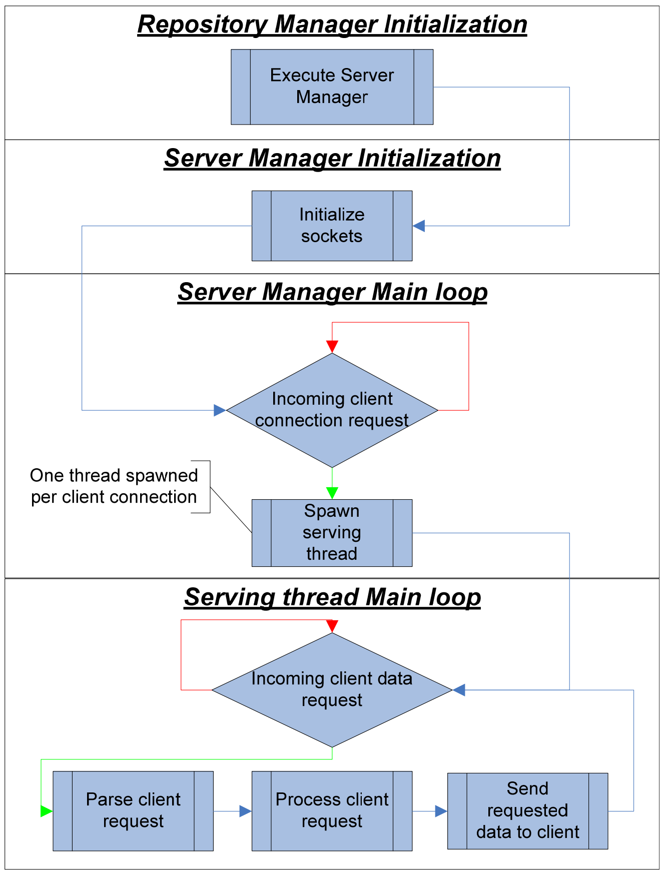

Repository Manager Initialization
Execute Server Manager
Server Manager Initialization
Initialize sockets
Server Manager Main loop
Incoming client connection request
One thread spawned per client connection
Spawn serving thread
Serving thread Main loop
Incoming client data request
Parse client request
Process client request
Send requested data to client

Figure 12 - Provider functional schema

2.3.1. Serving thread protocol

Keep in mind that this protocol won't be directly accessible from the clients since the *Client API* wrapper will hide it.

All requests and responses will be built by a header, and depending on the type of message, a body. The header will be common to all messages, containing a 4 byte signature (0x564F4944) in order to validate the message, a 4 byte code identifying the message type and the expected size in bytes of the body.

After each server response, a Ready message will be sent to confirm the server readiness to receive further requests, except for connection and disconnection messages.

The trace data server-client transferring will be the most bandwidth consuming. Because of this, the possibility of applying a compression layer to the data will be provided.
Several compression methods will be implemented (zip, FLAC and differential) and derived from benchmarking one or more will be retained in the final release version.

- **Managing connection**

- ***Connecting***

  Since there's no user authentication, the initial connection will be established by opening a socket between client and server. Once the socket link is up, the server will wait for client protocol acknowledgement header.

  Connection Header

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x434F4E4E
4 (int)	0

  When the server receives this header, it will be bounced back to the client to notify that connection was granted and it's ready to give response to client requests

- ***Disconnecting***

  Before closing the socket, the client should send a Disconnection header to allow the server to close the server-side connection gracefully.

  Disconnection Header

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x44495343
4 (int)	0

- **Client requests**

- ***Straight Channel list***

  Ask the server for a list of channels with available data in the server

Header

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x4C535449
4 (int)	0

- ***Metadata Channel list***

  Ask the server for a list of channels along their metadata

Header

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x4D455449
4 (int)	0

- ***Channel data***

  Ask the server for a single/multiple channel range of data

Header

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x44415449
4 (int)	8 + 20*N

Body

Bytes	Description
4 (int)	Channel count
4 (int)	Maximum packet data size
1 (char)	Compression flag
20 (struct)	Channel data request #1
...	...
20 (struct)	Channel data request #N

Channel Data Request structure

Bytes	Description
4 (int)	Channel ID
8 (double)	Start Time
8 (double)	End Time

- ✓ The channel ID will be one of those contained in the response to a Channel list request.
- ✓ Channel count will point how many channel info structures are present in the body.
- ✓ Max. packet data size will determine the Data field size limit in the response packets
- ✓ Compression flag: If enabled, requested data will be in compressed format. (0 = Uncompressed / n = Compression type)
- ✓ Starting and ending time will be in “seconds passed since 01/01/1970” format.

- **Server responses**

- ***Ready message***

Server is ready for dealing with the next request

Header

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x00524459
4 (int)	0

- ***Error message***

Server encountered an error while processing the current request

Header

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x00455252
4 (int)	8

Body

Bytes	Description
4 (int)	Last message type
4 (int)	Error code

✓ The Last message type will hold the sent message type value that generated the error

- ***Straight Channel list***

Return a list of available channels.

Header

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x4C53544F
4 (int)	4 + 48*N

Body

Bytes	Description
4 (int)	Channel count
48 (struct)	Channel Info #1
...	...
48 (struct)	Channel Info #N

✓ Channel count will point how many channel structures are present in the body.

Straight Channel Info structure

Bytes	Description
4 (int)	Channel ID
7 (NULL-terminated string)	Station
9 (NULL-terminated string)	Net
9 (NULL-terminated string)	Component
3 (NULL-terminated string)	Location
8 (double)	Start Time
8 (double)	End Time

- ✓ The channel ID will one assigned by the Repository Manager
- ✓ Starting and ending time will be in “seconds passed since 01/01/1970” format.

- ***<u>Metadata Channel list</u>***

Return a list of available channels along their metadata

Head

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x4D45544F
4 (int)	4 + 113*N

Body

Bytes	Description
4 (int)	Channel count
113 (struct)	Channel Meta #1
...	...
113 (struct)	Channel Meta #N

Metadata Channel Info structure

Bytes	Description
4 (int)	Channel ID
7 (NULL-terminated string)	Station
9 (NULL-terminated string)	Net
9 (NULL-terminated string)	Component
3 (NULL-terminated string)	Location
8 (double)	Start Time
8 (double)	End Time
8 (double)	Latitude
8 (double)	Longitude
8 (double)	Elevation
9 (NULL-terminated string)	Owner
8 (double)	Dip
8 (double)	Azimuth
8 (double)	Sample Rate
8 (double)	Sensitivity

- ***<u>Uncompressed Channel data</u>***

Head

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x4441554F
4 (int)	32 + N

Body

Bytes	Description
32+N (struct)	Channel Data

Channel Data struct

Bytes	Description
4 (short)	Channel ID
8 (double)	Start Time
8 (double)	End Time
8 (double)	Sample Rate
4 (int)	Number of samples
N (data)	Data

- ✓ The channel ID will one assigned by the Repository Manager
- ✓ Starting and ending time will be in “seconds passed since 01/01/1970” format.

- ***Compressed Channel data***

Head

Bytes	Description
4 (int)	0x564F4944
4 (int)	0x4441434F
4 (int)	N

Body

Bytes	Description
N (data)	Compressed Data

Receiving channel data:

Once a data request is processed by the server, it will begin sending Channel Data responses to the client. In a multi-channel data request the server will send the data for each channel sequentially, in the order pointed by the client in the data request message.

The Data member of the Channel Data structure in each response will be filled until any of these criteria is met:

- A trace data gap is found
- Data size limit specified by the client in its request is reached
- There’s a change in the data sample rate
- There’s no more data for current channel

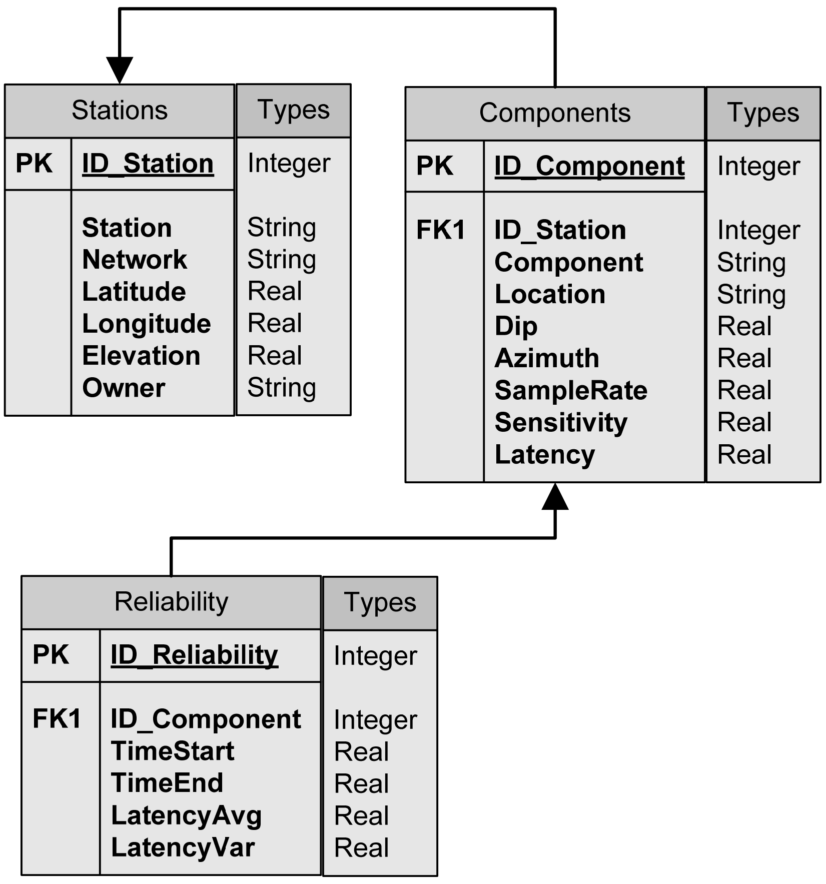
Every time one of these events happens and there are data left to be sent, a new Channel Data response will be generated following the same rules.

Only when all the required data from all the requested channels has been sent, the Ready message will be transmitted to the client and the data transfer will be considered concluded.

Received waveform data will be in raw format, with a resolution of 32bits per sample.

2.4. *Database*

- Database schema



Stations		Types
PK	**ID_Station**	Integer
	Station	String
	Network	String
	Latitude	Real
	Longitude	Real
	Elevation	Real
	Owner	String

Components		Types
PK	**ID_Component**	Integer
FK1	**ID_Station**	Integer
	Component	String
	Location	String
	Dip	Real
	Azimuth	Real
	SampleRate	Real
	Sensitivity	Real
	Latency	Real

Reliability		Types
PK	**ID_Reliability**	Integer
FK1	**ID_Component**	Integer
	TimeStart	Real
	TimeEnd	Real
	LatencyAvg	Real
	LatencyVar	Real

Figure 13 - Database tables' schema

- Relationship between the tables

Stations	1	→	n	*Components*
Components	1	→	n	*Reliability*

Figure 14 - Database tables' relationships

- Primary keys

  All primary keys will have auto incremental values. In Oracle there aren't auto fields, so a SEQUENCE database object will be used for each primary key in order to generate unique values.

Every time a new register is inserted, a new primary key will be generated with the Oracle/SQL sentence "… Sequence_name.NextVal() …"

Stations table

This table will contain the fields describing the metadata of a station and part of its SCNL code. Each inserted register represents a station.

Field	Type	Required	Details
ID_Station	NUMBER(13)	Yes	Primary key
Station	VARCHAR2(7)	Yes	The S on SCNL
Network	VARCHAR2(9)	Yes	The N on SCNL
Latitude	NUMBER(10,6)	No	Station's latitude for its geographical location, stored in degrees
Longitude	NUMBER(10,6)	No	Station's longitude for its geographical location, stored in degrees
Elevation	NUMBER(7,2)	No	Station's elevation above sea level, stored in meters
Owner	VARCHAR2(16)	No	String identifying the station's owner

Table 3 - Fields of stations table containing metadata of each station and part of its SCNL code

Constraints

Station and *Network* fields are paired to form a unique constraint which defines stations' individuality, so no duplicate pairs will be allowed.

Oracle DDL sentences

```
CREATE SEQUENCE Seq_ID_Station INCREMENT BY 1 START WITH 1;

CREATE TABLE Stations     (        ID_Station      NUMBER        (13),
                                   Station         VARCHAR2      (7) NOT NULL,
                                   Network         VARCHAR2      (9) NOT NULL,
                                   Latitude        NUMBER        (10,6),
                                   Longitude       NUMBER        (10,6),
                                   Elevation       NUMBER        (7,2),
                                   Owner           VARCHAR2      (16)
                          );

ALTER TABLE Stations ADD CONSTRAINT PK_Station PRIMARY KEY (ID_Station);

ALTER TABLE Stations ADD CONSTRAINT  UK_Station UNIQUE (Station, Network);
```

Components table

Registered components from existing stations will be stored in this table.
Each inserted register represents a station's component.

Field	Type	Required	Details
ID_Component	NUMBER(13)	Yes	Primary key
ID_Station	NUMBER(13)	Yes	Foreign key to Stations table
Component	VARCHAR2(9)	Yes	The C on SCNL
Location	VARCHAR2(3)	No	The L on SCNL
Dip	NUMBER(4,1)	No	Component's inclination measured in degrees
Azimuth	NUMBER(4,1)	No	Component's azimuth measured in degrees
SampleRate	NUMBER(4,1)	No	Component's frequency measured in Hz
Sensitivity	NUMBER(7,2)	No	Component's sensitivity in counts per measuring unit at 1 Hz
Latency	NUMBER(6,6)	No	Component's most current data transmission latency measured in seconds

Table 4 - Components table fields' description

Constraints

Component and *Location* fields are paired to form a UNIQUE constraint which defines components' individuality (along with the foreign key), so no duplicate pairs will be allowed.

Oracle DDL sentences

```
CREATE SEQUENCE Seq_ID_Component INCREMENT BY 1 START WITH 1;

CREATE TABLE Components (         ID_Component   NUMBER       (13),
                                  ID_Station     NUMBER       (13) NOT NULL,
                                  Component      VARCHAR2     (9) NOT NULL,
                                  Location       VARCHAR2     (3),
                                  Dip            NUMBER       (4,1),
                                  Azimuth        NUMBER       (4,1),
                                  SampleRate     NUMBER       (4,1),
                                  Sensitivity    NUMBER       (7,2),
                                  Latency        NUMBER       (10,6)
                         );

ALTER TABLE Components ADD CONSTRAINT PK_Component PRIMARY KEY (ID_Component);

ALTER TABLE Components ADD CONSTRAINT FK_Component FOREIGN KEY (ID_Station)
REFERENCES Stations (ID_Station);

ALTER TABLE Components ADD CONSTRAINT  UK_Component UNIQUE (Component, Location,
ID_Station);
```

Reliability table

Here will be saved the information regarding component's stored data reliability. Each inserted register represents an uninterrupted component's trace data sequence.

Ideally, there should be a unique register for each unique component, as that would mean the feed has been constant and gapless.

On the other hand, the presence of several registers corresponding to one unique component implies the existence of gaps, being the number of gaps equal to the number of registers minus one. The absolute difference between the ending time from register A and the starting time from register B (where B is the register that follows in time to A) will give the gap time length.

Field	Type	Required	Details
ID_Reliability	NUMBER(13)	Yes	Primary key
ID_Component	NUMBER(13)	Yes	Foreign key to Components table
TimeStart	NUMBER(14,4)	No	Data sequence starting time, stored in seconds since 00:00 01/01/1970
TimeEnd	NUMBER(14,4)	No	Data sequence starting time, stored in seconds since 00:00 01/01/1970
LatencyAvg	NUMBER(6,6)	No	Computed average latency from the packets which built the sequence
LatencyVar	NUMBER(6,6)	No	Variance from the latency average

Table 5 - Reliability table fields' description

Oracle DDL sentences

```
CREATE SEQUENCE Seq_ID_Reliability INCREMENT BY 1 START WITH 1;

CREATE TABLE Reliability     (       ID_Reliability   NUMBER      (13),
                                     ID_Component     NUMBER      (13) NOT NULL,
                                     TimeStart        NUMBER      (14,4),
                                     TimeEnd          NUMBER      (14,4),
                                     LatencyAvg       NUMBER      (6,6),
                                     LatencyVar       NUMBER      (6,6)
                             );

ALTER TABLE Reliability ADD CONSTRAINT PK_Reliability PRIMARY KEY (ID_Reliability);

ALTER TABLE Reliability ADD CONSTRAINT FK_Reliability FOREIGN KEY (ID_Component)
REFERENCES Components (ID_Component);
```

2.5. *Administration web tool*

- Authentication

Being a tool designed to modify the database contents, a security layer will be applied. Login and password will be required before further interaction can be achieved. Implementation of authentication will be the one imposed by the chosen web server: Apache2.

- Interface

Once the user has been authorized by entering a valid login and password, a table containing the list of already inserted stations and their metadata will be displayed, ordered alphabetically by station code.

- Interaction: Adding a station

User will be able to insert a new station. She must provide input for all the required fields present in Table 3.

When submitting the filled fields a client-side validation process will be triggered:

- Floating point conversion. The decimal separator character will be a decimal point.

- All fields will be checked to verify conformance with the expected types.

- Validate database constraints: The fields Station and Network pair from the new station mustn't be present in the database.

After successful validation, the web server will establish a connection to the database and will insert the new station along its metadata into the stations table. Once committed, the web page will refresh showing the list with the newly included station.

- Interaction: Editing a station

User will be able to edit an existing station. Editing will be restricted to non-constrained fields:

- Latitude
- Longitude
- Elevation
- Owner

If the user wants to edit the constrained fields (Station and Network) she will have to follow a "manual" process: Deleting current register and creating a new register with the Station and Network desired values along the metadata.

Validation will occur like described in "Adding a station" section. After successful validation, the web server will establish a connection to the database and will update the station along its metadata into the Stations table.

The web page will refresh showing the list with the newly included station.

- Interaction: Deleting a station

User will be able to delete an existing station.
A confirmation message will be displayed before permanent deletion is completed.

Deleting a station will operate in cascade mode, removing any associated components. This is done to keep the consistency in the relational database structure.

- Interaction: Adding a component

User will be able to add a station component. . She must provide input for all the required fields present in the components table (Table 4), except latency which will default to NULL value.

An existing station is needed as it will be the owner of the component and the database design doesn't allow orphan component entries. The interface will restrict the addition of components to fit the above statement.

Validation will occur like described in "Adding a station" section.

After successful validation, the web server will establish a connection to the database and will insert the new component along its metadata into the Components table.

The web page will refresh showing the list with the newly included component.

- Interaction: Editing a component

User will be able to edit an existing component.
Editing will be restricted to non-constrained fields:

- Dip
- Azimuth
- Sample rate
- Sensitivity

After successful validation, the web server will establish a connection to the database and will update the component along its metadata into the Components table.
The web page will refresh showing the list with the newly included component.

- Interaction: Deleting a component

User will be able to delete an existing station component. A confirmation message will be displayed before permanent deletion is completed.

The web page will refresh showing the list with the newly included component.

2.6. *Monitoring web tool*

- .Authentication

The *Monitoring tool* will only execute read operations, so there's no database integrity risk. Therefore, an authentication system isn't needed in this case.
No user interaction will be present in this tool, besides reviewing the displayed information.

- Interface

Basically, the *Monitoring tool* is a table showing all the established feed links, ordered alphabetically by Station code. It will connect periodically to the database to read the latest available data for each feed from the *Reliability table*.

The following parameters will be shown:

- Station code (string)
- Network code (string)
- Component code (string)
- Location code (string)
- Available data starting time (real)
- Available data ending time (real)
- Channel merged average latency (real)

- Merged Standard deviation for average latency (real)
- Channel most current average latency (real)
- Data availability (graphic bar)

Station code	Network code	Component code	Location code	Starting time	Ending time	Merged Average latency	Merged Standard deviation latency	Current average latency	Data availability

Table 6 - Available fields of web monitoring tool

Data availability will be a color-coded bar, showing in green the available data, in red the absence of data (gaps).

The graphical bar will be attained by under-scaling component's time range to fit an integer range, from *0* to *n*, with a resultant time unit resolution of (*Ending Time - Starting Time) / n*.

The loss of time native resolution implies that there's a minimum gap length threshold under which a gap won't be noticeable. To address this issue a discontinuity display line will be implemented, in the form of a yellow line over the bar that will break every time a gap is encountered, no mattering its size.

- Implementation

In order to display the required information, the monitor tool will have to access to *Stations, Components* and *Reliability* database tables, so it can build each feed information line which will result in a list with components reliability data that can be viewed as follows:

Sta	Net	Comp	Loc	Latency	TimeStart	TimeEnd	LatAvg	LatVar
CAVN	CA	HHZ	--	3,863962	1268735449	1268738866	4,02005	1,242655
					1268738869	1268741251	3,96197	1,137046
		HHN	--	3,213374	1268745927	1268747059	3,26969	1,004475
					1268747062	1268747669	3,04733	,9880953
					1268747672	1268748329	3,55418	1,020006
					1268748332	1268749143	3,18140	1,003208
		HHE	--	2,845733	1268749201	1268749362	2,82012	1,066519
					1268749365	1268749583	2,88440	,9927881
CFON	CA	HHZ	--	4,127426	1268746163	1268749580	4,23330	,9294196
					1268746164	1268749581	3,95122	,9654599
		HHE	--	8,347942	1268746163	1268749580	8,15502	2,706073

Table 7 - Components information fiels and example of data obtained by monitoring tool from de database

- Starting and ending time for the component will be the minimum TimeStart and maximum TimeEnd values from all the sequences for that component.

- To obtain the merged average latency and merged standard deviation latency, the latency average and standard deviation of all the sequences for that component must be accumulated using the following algorithm:

Knowing that…

…Sequence A contains…
Latency Average, from now on, La
Latency Stddev from now on, Sa
Time interval, from now on, Ta

…Sequence B contains…
Latency Average, from now on, Lb
Latency Stddev, from now on, Sb
Time interval, from now on, Tb

…Merged Sequence C will contain…
Merged latency Average, from now on, Lc
Merged latency Stddev, from now on, Sc
Merged Time interval, from now on, Tc

Merged Sequence C components will be computed as…

Tc = Ta + Tb
Lc = ((Ta * La) + (Tb * Lb)) / Tc
Sc = sqrt((Ta*(Sa2) + Tb*(Sb2) + Ta*(La2) + Tb*(Lb2) - Tc*(Lc2)) / Tc)

2.7. *The sample client*

Sample client is a lineal process with an input and an output that can be detailed as follows:

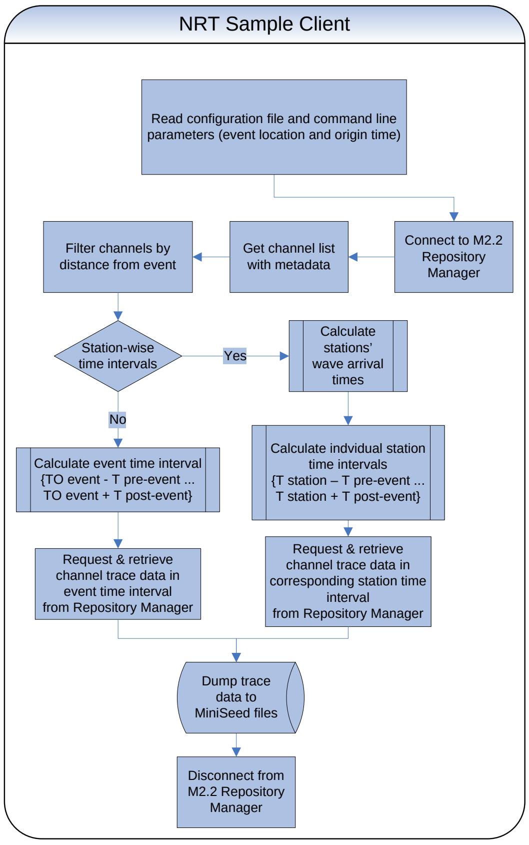

NRT Sample Client
Read configuration file and command line parameters (event location and origin time)
Connect to M2.2 Repository Manager
Get channel list with metadata
Filter channels by distance from event
Station-wise time intervals
Yes
Calculate stations' wave arrival times
No
Calculate event time interval {TO event - T pre-event ... TO event + T post-event}
Calculate indvidual station time intervals {T station – T pre-event ... T station + T post-event}
Request & retrieve channel trace data in event time interval from Repository Manager
Request & retrieve channel trace data in corresponding station time interval from Repository Manager
Dump trace data to MiniSeed files
Disconnect from M2.2 Repository Manager

Figure 15 - NRT Sample client program flow

- Running the client

When client execution begins, the first task is to read the configuration file, found in the binary directory, and to read the event location and origin time passed by command line parameter.

From the configuration file the client will obtain:

- Distance (radius in km.) that will be used when calculating stations inside a defined circular area.
- Pre-event and post-event time modifiers (in seconds) to be used in calculation of channel's data time intervals.
- Mini-seed naming convention and storage path.
- NRT server address and connection parameters.
- Log file path.
- Channel filtering using wildcards that will restrict channels used by this client.
- Flag to enable/disable station-wise intervals.

From the command line parameters will obtain:

- Event latitude and longitude in degrees
- Event depth in meters
- Event origin time in year, month, day, hours, minutes and seconds.

- Connect to Repository Manager

Once the client has processed the configuration file and command line parameters will use the connection parameters read from the configuration file to connect to the *NRT server* using the *Client API*.

- Getting channel list

After connection has been established the client will ask for an updated list of available channels. This list will include the metadata for those channels, which will be used in the next step.

- Filtering the channel list

The client must determine which channels are near enough to the event geographical coordinates.

Flagging valid stations will be done using the stations location, event location and the distance constant read from the configuration file. Any channel owned by a station laying inside the circumference centred at the event location with radius equal to the distance constant, will be included in the filtered channel list.

When the filtered list is ready, it will be matched against the filtering wildcards read from the configuration file to provide the final channel list.

- Event-wise and station-wise intervals

Having the event time origin and the list of nearby stations, it's time to request the trace data from the channels. To request the data a time interval is necessary, and this interval can be calculated using two different references, depending on the flag specified in the configuration file.

- The path of the event-wise interval

  The client will use the event origin time, and by adding and subtracting the post and pre-event constants respectively, obtained from the configuration file, it will obtain the event-wise interval.
  Then it will use this interval as a common value for all channel trace data requests.

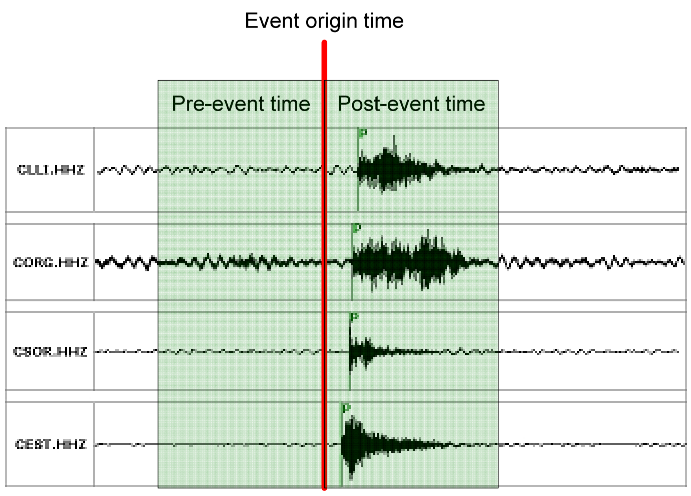


Figure 16 - Sample client: graphical example of event-wise interval calculation

- The path of the station-wise interval

  Instead of using a common interval for all data requests, an individual interval will be calculated for each channel.

  With the station's wave arrival times, the method is the same as in event-wise interval. Instead of using event origin time as the reference for the interval calculation, client will use the arrival times to calculate individual intervals.

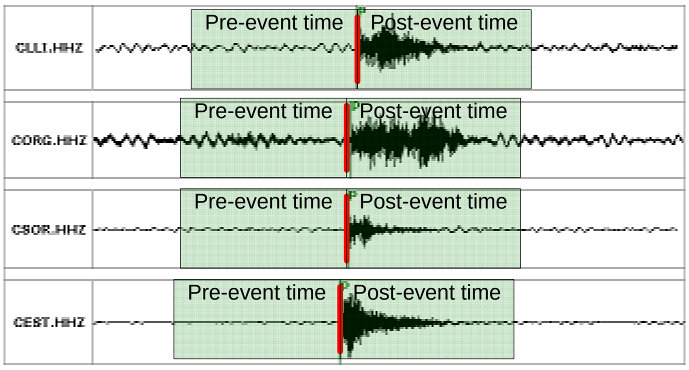


Figure 17 - Sample client: graphical example of station-wise interval calculation

- Getting the trace data

By using the *Client API* the client will call the requesting procedure to get the trace data for each channel sequentially. *Client API* works over a blocking socket, so after a channel data request, the client will have to wait till the data is retrieved or a timeout expires.

- Exporting to MiniSeed

Using the path and naming convention found in the configuration file the client will convert the trace data received from *NRT server* to MiniSeed format and dump it to disk.

3. System test

Pools of test have been carried out in order to check the performance of the system. A summary of these tests is described at this section.

3.1. *Test scenarios*

3.1.1. Basic scenario

- Requester scenarios
  - Multi Feeding scenarios
    - 2..N NAQS servers feeding n channels to n tanks
    - 2..M Seedlink servers feeding n channels to n tanks
    - 2...P Scream! servers feeding n channels to n tanks
  - Complex Feeding scenarios
    - Combination of NAQS, Seedlink and Scream! servers feeding Q tanks

  All these scenarios include:

  - Station and component information registered into Oracle DB through Administration tool
  - Reliability data stored on Oracle DB
  - Checking feeds reliability using the Monitoring tool

- Provider scenarios
  - Single simple client connection (Windows version only)
  - Multiple simple clients connection (Windows version only)
    - ✓ Simple client: Client with reduced functionality, having only Connect, Channel List request, Channel Data request and Disconnect messages implemented.

  These scenarios include:

  - Connection and handshaking. Disconnection.
  - Channel list requesting
  - Metadata requesting
  - Data requesting

3.1.2. Full scenario

Same conditions as in Basic scenario, but replacing simple client with the sample client described in Design documents, both Linux and Windows versions. Also implies testing of concurrent access to Oracle database and tanks.

3.2. *Test cases*

Common channel's issues for all cases

- Different sample rates between several channels
- Different sample rates in the same channel
- Presence of gaps
- Channels with different data types
- Data retransmission

3.2.1. Case 1: Standard error-free environment

It will be defined as an environment without on purpose generated errors.

3.2.2. Case 2: Connectivity issues

- Case 2.1: Broken feed link
    - Case 2.1.1: Station to datacenter (no data)
    - Case 2.1.2: Datacenter to NRT server (no connection)
- Case 2.2: Broken Oracle Database link
- Case 2.3: Broken Apache Web server link
- Case 2.4: Client wrong implementation (Data protocol testing)

3.2.3. Case 3: Storage issues

- Case 3.1: Database corruption
    - Lost/unreal/extreme values for reliability data
    - Lost/unreal/extreme values for component data
    - Lost/unreal/extreme values for station data
- Case 3.2: Tank corruption
    - Case 3.2.1: Tank file corruption / erase
    - Case 3.2.2: Index file corruption / erase

  - Case 3.2.3: Structure file corruption / erase
- Case 3.3: Testing of crash recovery mechanisms

3.2.4. <u>Case 4: Stressing</u>

- Case 4.1: Connection-wise
  - Case 4.1.1: Simultaneous Feed links
    - Case 4.1.1.1: From same datacenter source
    - Case 4.1.1.2: From different datacenters
  - Case 4.1.2: Client connectivity concurrency
  - Case 4.1.3: Hammering
    - Case 4.1.3.1: Database & Web server, through Monitor tool
    - Case 4.1.3.2: Tanks, through client applications
- Case 4.2: Storage-wise
  - Case 4.2.1: Number of tanks
  - Case 4.2.2: Tank's parameters
- Case 4.3: Server

Study of resource consumption

4. Operation

Nowadays 51 stations (153 streams) are being received at NRT Server installed and in operation at IGC facilities in Barcelona. All these data are available for partners and for the shake maps generation.

NRT	IGN	BRGM	OMP	IGC	TOTAL
ACC	2	9	6	6	23
BB	7	0	7	13+1 (IAE)	28
TOTAL	9	9	13	20	**51**

Table 8 - Distribution of BB and accelerometric stations received at NRT Server, from each partner.

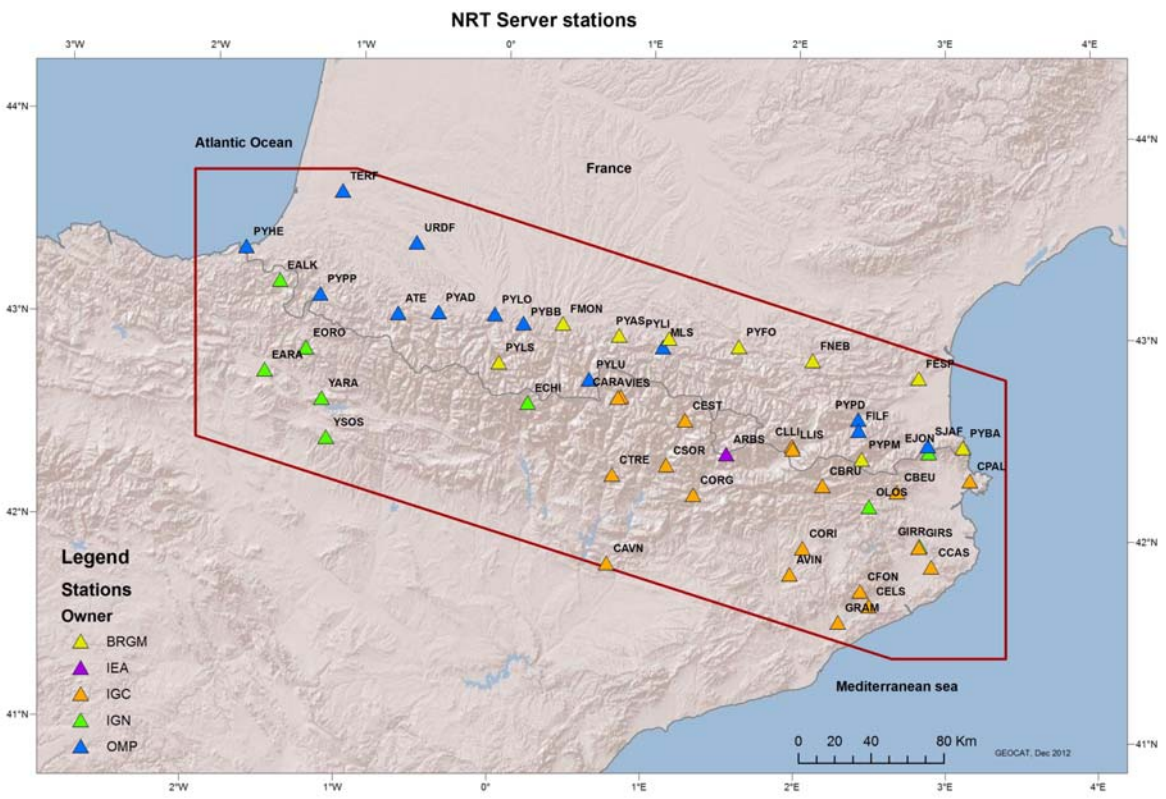


Figure 18 - Map of all stations received at the NRT Server, identified by their owner. The red polygon represents the SISPyr region.