

# License

Copyright © 2004, Wade Tregaskis. All rights reserved.

Redistribution and use in any forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions must retain the above copyright notice, this list of conditions and the following disclaimer.

- \* Neither the name of Wade Tregaskis nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS DOCUMENT IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# Mission to the ISS

# Server Architecture Specification

v1.0b – Wade Tregaskis

## Overview

This document details the server architecture in broad terms. It does not concern itself with pedantic details like class definitions or hierarchies, but rather defines the methodologies adopted, the overall structure and the processes involved.

For a documentation of the classes used and similar such details, refer to <insert references here>.

## Approach

The server is designed to be largely platform agnostic – a simple POSIX CLI application. It does not use any technologies (libraries or other functionality) not available on *all* the target platforms, which at present stand at Windows and \*nix (encompassing Linux, the BSDs and MacOS X). This is all fairly straightforward, as the server has little need to use outside libraries and functionality beyond defacto standards (e.g. STL, stdc, POSIX sockets, etc).

## Functionality

To understand the approach taken to the server design, it is necessary to define at minimum it's required functionality. This is the minimum set of actions the server must be capable of performing. They are:

1. Generate suitable data for all systems involved in the simulation. In particular, most individual data sets are related to other data sets (e.g. ionising radiation levels are generally proportional to magnetosphere strength). Consequently the generated data must not only fit appropriate statistical distributions, but also must meet the logical requirements and bindings as exemplified above.
2. Push data to all client systems at appropriate times. The primary requirement here is real time operation, with communication of historical data a secondary aim. Latencies must be minimal (tens of milliseconds) between the occurrence of data and it's display by clients.
3. Manage client input into the system, and perform appropriate validation of that input. Clients are able to manipulate parameters of the simulation (e.g. power levels to various systems) and need to be able to apply these manipulations in a timely fashion. Subsequently, the server needs to adapt it's data generation appropriately given any manipulations.
4. Record all data used in the simulation for later analysis. The ability to perform a postmortem of the simulation may be an essential learning tool – i.e. being able to take students back to particular points and explain what was really happening (possibly unbeknown to them) and what should have been done, perhaps as opposed to what was done.
5. Schedule pre-programmed anomalies or other such events. These would be parameterised in a minimal fashion (e.g. anomaly “solar flare” at time xxxx, peak strength = zzzz), and would be applied in a suitable fashion [at the appropriate time] to the data generators. There is no primary requirement that whole sets of data need be specified.

In addition to the required functionality, there are many features that are desirable, if they can be implemented within the given deadlines. They include:

1. Replay existing data sources, as opposed to generating them in real time statistically. This might be used to feed real-world data into the system, or perhaps to retry a previous simulation, with the ability to try a different course of action at some critical point.

In addition, partial replay would be useful for applying precise anomalies or similar – e.g. inserting a set of pre-generated data into some future part of the simulation. This acts as an extension of the pre-defined anomaly generators, allowing for exact specification of raw data, thus allowing any activity to be simulated, not just the types pre-determined.

2. Pull real time data from NASA and other such websites. For example, the actual position of the ISS could be read and continuously updated, with students required to plan the shuttle launch and flight to appropriately meet up with the ISS as it orbits. This may be limited, however, by practical limitations of the simulation – for example, it may be that the launch window for an ISS-bound shuttle is only a few minutes every few days, in which case it is not practical to require students to “come back” to the simulation later, when it’s ready.

A more practical aspect of this might be imagery, for example of the Earth and the Sun. These are not so critically tied into simulation data, and so there are fewer concerns with them being suitable for the simulation.

3. Generate appropriate random anomalies, confined suitably by any real-time or pre-configured data. The simulation needs to be at some level aware that it needs to maintain a challenge for all the people involved, and so needs to feed appropriate stimulus (in the form of anomalous events) to all clients at appropriate times and given appropriate intervals.

Given the above requirements, we can finally begin to consider the server architecture. At a fundamental level, it acts as TCP host to a number of clients (possibly a very large number, if “observer” clients are supported). At a minimum it needs to support 2 clients for the purposes of the major project, plus a third administrator client as part of the minor project (which will induce an effective workload of at least 2 normal clients).

There are three typical architectures for the server, in terms of client communication. They are:

- Per-client Process – For each client a new process is spawned, in much the same fashion as Apache and similar servers operate. Pros are that there are few synchronisation issues, and that the overall system is slightly more stable as one process crashing effects only one client. Cons are substantial for our purposes – ultimately there must exist a single controlling server process, as all data must be centrally and singularly co-ordinated. So really this architecture simply introduces an extra layer of network complexity, which is largely redundant.
- Per-client Thread – For each client a new thread is spawned. Pros are that since everything remains in a single process, data sharing is efficient and trivial, although there exist significant synchronisation issues. Additionally, the server will scale better on multi-processor machines than a serialised server, which may be an important factor when a full 12-client-plus system is considered. Cons include a reliance on a single process, which requires more robustness than a per-client process architecture. Additionally, there is a need for careful synchronisation and resolution of other concurrency issues, which introduces not only a significant runtime overhead, but a significant level of complexity to the programming.
- Serialised – The server runs as a single serial thread, shifting it’s “focus” between clients as necessary (i.e. using select). Pros include simplicity and safety with regards to concurrency issues (as in the per-client thread architecture). Cons include a lack of scalability on multi-processor machines, plus potential inefficiencies with regards to resource management – e.g. network time may be wasted while the server performs a CPU-intensive task, introducing both higher latencies in client/server communication and potential bottle-necks if the amount of data being communicated becomes significant.

It cannot be avoided that if real-time data is to be acquired from outside sources, a serialised model is simply impractical – the lack of prioritisation between acquisition of [possibly non-essential] data and the normal management of the simulation will inevitably lead to problems.

**This move to adopt a per-client process model (or close hybrid there-of) is at time of writing yet to be ratified by the other relevant group members.**

The proposed TCP architecture is to use a lightweight class wrapper over POSIX sockets, which will provide for blocking IO with the option to interrupt idle operation in order to transmit data. Meaning, the thread will be able to block on the socket waiting for data – the most efficient way of working with sockets – but can be interrupted, to allow the thread to re-check it's send buffers for any new data submitted by other threads.

In this way each client thread will remain idle unless processing data from the client, or transmitting data to the client. There are no significant latencies introduced at any point, aside from when both client and server wish to transmit data – in such a case it's a bit of pot luck as to what happens first and who gets their request handled first. This should be irrelevant, however, as no one task should occupy the client thread for any significant amount of time; all significant tasks will be offloaded to so-called “worker” threads (e.g. major requests for archived data).

From the point of view of the aforementioned worker threads, if they wish to transmit data to a client they will need to append record of that data to the relevant socket's outgoing buffer. This “buffer” will in fact be composed of a mutex-controlled linked list of simple structures, which will be able to specify either an inline data buffer or refer to some outside source. The latter is useful because for large amounts of immutable data – e.g. historical data for any particular measurement or instrument – there is no need to copy the data to an intermediate buffer prior; it is already read only, and so there are no synchronisation issues or race conditions. Having placed all relevant records into the buffer, the worker then interrupts the socket via a special mechanism (details not relevant here), which will cause the client thread to step back and check it's outgoing buffer. Finding the new data there, it will then begin transmission of that data.

It should be noted that no such complexity is necessary on the client side – each individual client application (representing one mission control terminal) needs only a single TCP socket connection to the server, and so should be able to efficiently manage that without such elaborate measures as detailed above. In addition, I understand Borland provides efficient asynchronous operation on sockets.

Note that asynchronous operation is possible with POSIX sockets, but it requires the use of signals to notify the process of socket events. Signals are by no means sluggish, but awkward to deal with in a complex environment – especially with many sockets – and unlikely to perform anywhere near as well as the previously proposed approach.

## Execution Path

As the server utilised threads heavily for performance and concurrency reasons, there is no linear execution path. Consequently an introductory overview of the execution paths is necessary.

Before that, however, it's necessary to specify exactly how the server spawns new threads. It initially sets it's signal mask to block all signals. It then spawns any threads it needs (as will be covered shortly). These new threads inherit the mask of all signals blocked. Once all “1<sup>st</sup> generation” (those launched from the main thread) threads have been spawn, the main thread removes the signal mask. In this way the main thread is the only thread that can receive signals, which aids in graceful shutdown (as covered later).

When the server launches it performs all it's initialisation (including setting the signal mask, discussed previously), and much of it's configuration loading, such as reading in simulation parameters and so forth. This is prior to calling daemon() (if operating as a daemon) or providing interactivity (if operating interactively). If initialisation and configuration is successful, the server either:

- a) Spawns a new thread for user interaction via the command line.
- b) Calls daemon() to detach from the terminal and operate entirely in the background.

In either case, it then begins to execute each Simulation. For each Simulation it spawns a new thread to execute that simulation. Once all Simulation's have been executed (in their separate threads) the main thread then unblocks all signals and calls sigsuspend() in an infinite loop, allowing the main thread to accept so-called "fatal" signals and perform a graceful shutdown.

Each Simulation operates one or more listeners that receive incoming networking connections. Each one of these is approved or disapproved by the ClientManager (a helper class for the Simulation) based on connecting IP or other influences. If approved, the ClientManager creates a new Client instance and executes it in a new thread. All client interaction is then performed on that new thread, leaving the ClientManager's thread free to continue performing it's duties.

So, each Simulation's main thread (not to be confused with "the" main thread, i.e. the program's main thread) spends most of it's time in the ClientManager, waiting for connections. Additionally, each Simulation may spawn any number of "worker" threads at appropriate times (e.g. when the simulation begins). These worker threads may perform a variety of tasks, primarily concerned with generating simulation data and managing inputs to the simulation (e.g. from clients).

When a client disconnects, the Client instance that managed it exits, and the thread upon which it was running terminates. When the server is instructed to shut down (whether interactively, by remote administration or from an appropriate Unix signal) it instructs each Simulation to close. This will initially shut down the ClientManager (and it's listeners), and then the worker threads (if any). The main thread itself never exits – once shutdown is complete, exit() is called forcefully.

A summary flowchart of all this is shown below.

