Bus Tracker Performance Architecture
“Model” Solution

Ewan Tempero

Architecture Description

Module structures

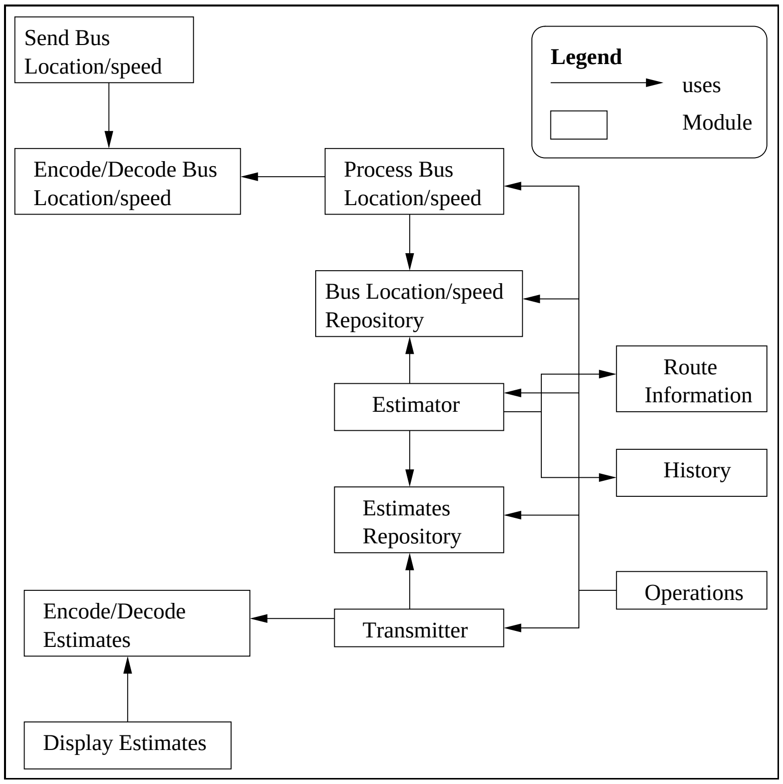

Figure 1: “Uses” Structure for BusTracker

Figure 1 shows the “uses” structure for the BusTracker architecture.

The **Send Bus Location/speed** module is responsible for collecting the information from the GPS system and packaging and sending that information via the bus transmitter. It uses the **Encode/Decode Bus Location/speed** module to do the packaging.

The **Process Bus Location/speed** module is responsible for dealing with the packets received at the radio receiver, in particular unpacking them (also using the **Encode/Decode Bus Location/speed** module and doing any redundancy and consistency checking required. It uses the **Bus Location/speed repository** to store the information it has received.

The **Bus Location/speed repository** module stores, for each bus, it’s current location and speed. When a new location and speed is supplied by the **Process Bus Location/speed** module, the old ones are discarded. This module also keeps track of new buses that show up.

The **Estimator** module takes the current location and speed of a bus, along with information about the route that it’s on and any relevant historical information to create estimates of its arrival times at the major bus stops during the next hour. It stores the resulting estimates using the **Estimates Repository**. It also passes the location and speed information to the **History** module, to update the history records.

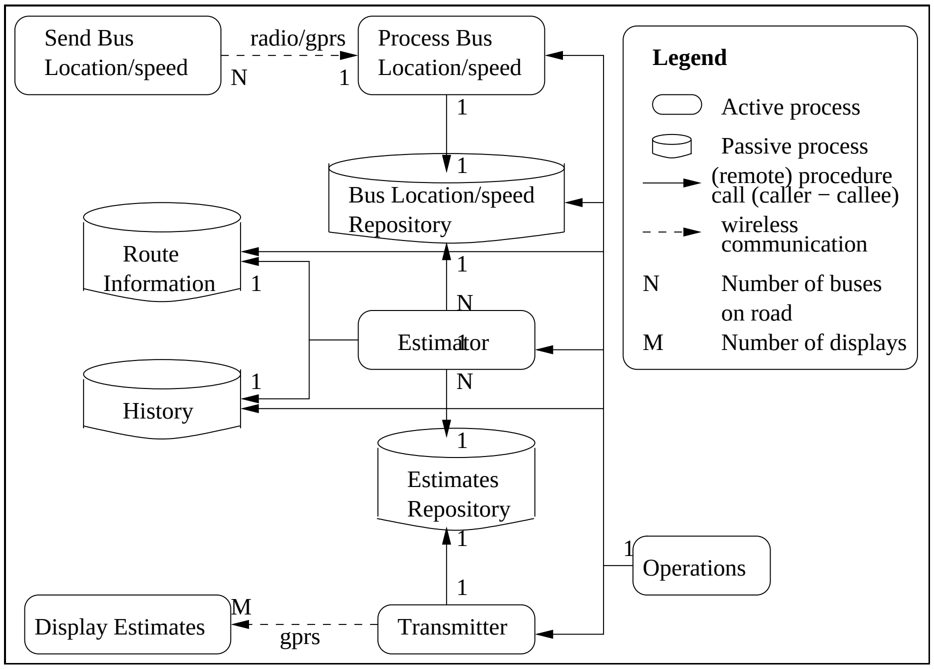

Figure 2: "Process" Structure for BusTracker

The **History** and **Route Information** modules are responsible for providing history and route information in appropriate forms.

The **Estimates Repository** stores, for each (display,bus) combination, the expected arrival time for that bus at that display (or nothing if the bus is not estimated or scheduled to arrive within the next hour). Whenever a new estimate is supplied by the **Estimator** module, the old estimate is discarded.

The **Transmitter** module is responsible for taking the estimates stored in the **Estimates Repository** and packaging them up and sending them to the displays. It uses the **Encode/Decode Estimates** module to do the packaging.

The **Display Estimates** module runs on the display subsystems. It receives and unpackages the estimates (also using the **Encode/Decode Estimates** module) and displays them.

The **Operations** module is responsible for overall control and monitoring of the system. It provides a user interface to the humans to allow all aspects of the central control system to be monitored, as well as updating route and history information. In particular, it monitors **Bus Location/speed Repository** to determine when new buses appear, and controls when estimates start being generated for them. Finally, it is the **Operations** module that initiates the sending of estimates to the display.

Component and Connector structures

Figure 2 shows the process structure for the BusTracker architecture. There are two different types of components: "active" processes and "passive" processes. Active processes run independently from all other components and can initiate communication, in the same way (for example) a Java Thread can. Passive processes cannot initiate communication; they can merely respond to requests. They could be regarded a "servers", and so be in a client/server relationship with active processes, but they may not exist a real processes, depending upon how they are deployed (see below).

All processes in the system correspond to a module (although there are a couple of modules that aren't also processes), and so the same names are used for both the modules and processes. There are as many **Send Bus Location/speed** processes as there are buses and as many **Display Estimates** processes as there are displays. The number of **Process Bus Location/speed** processes is currently unspecified — see the discussion below. There are as many **Estimator** processes as there are buses, and there is one each of **Transmitter** and **Operations** processes.

All wireless communication is asynchronous and all other communication is synchronous.

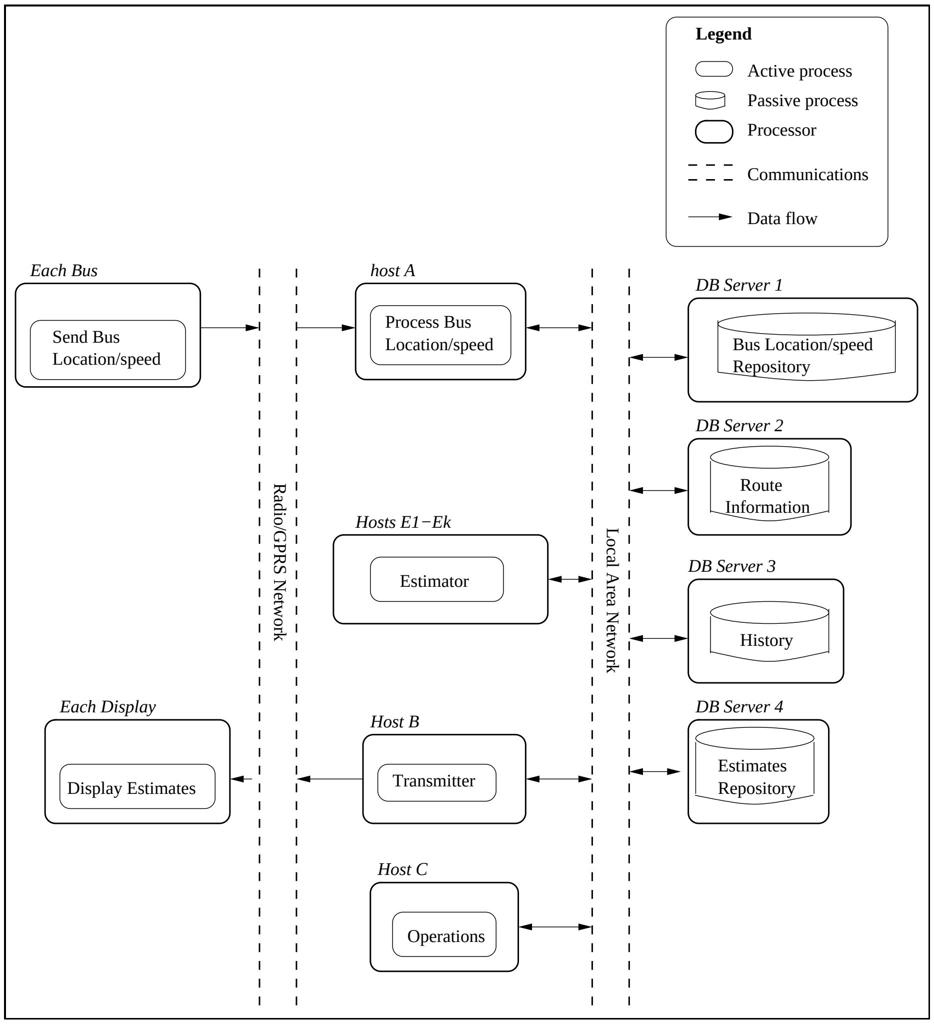

Figure 3: “Deployment” Structure — Many processors

Allocation Structures

Figures 3 and 4 show two possible deployment structures showing how processes can be allocated to processors. Roughly speaking, they show two extremes. In both, the **Send Bus Location/speed** processes run on each bus subsystem (one per subsystem) and the **Display Estimate** processes run on each display subsystem. In Figure 3, each of the **Process Bus Location/speed**, **Transmitter**, and **Operations** processes to be run on separate processors, but in Figure 4, they are all run together on the same processor. Similarly, the passive processes could be run on separate database servers, or run together on one. Variations in between are also possible. For example, the **Transmitter** and **Operations** processes could be run on the same processor. The final decision for that can be delayed until more real performance data is gathered.

Given the lack of performance information on the estimation algorithm, this architecture assumes that the **Estimator** processes will run on multiple processors (in the figures, k processors are used). The exact number will depend on the algorithm performance, but conceivably it could be one process per processor.

Other variations are also possible. The passive processes can be co-located with one or several of the active processes. For example, Figure 5 shows a view of the deployment structure showing the **Bus Location/speed Repository** process being run on the same processor as the **Process Bus Location/speed** process, and the **Estimates Repository** process run on the same processor as the **Transmitter** process.

There are some decisions that can be changed (discussed in the Justification Section below).

Tactics

The following performance tactics are, or could be, used in this architecture. More detail as to how some of these tactics are used is given in the Justification Section below.

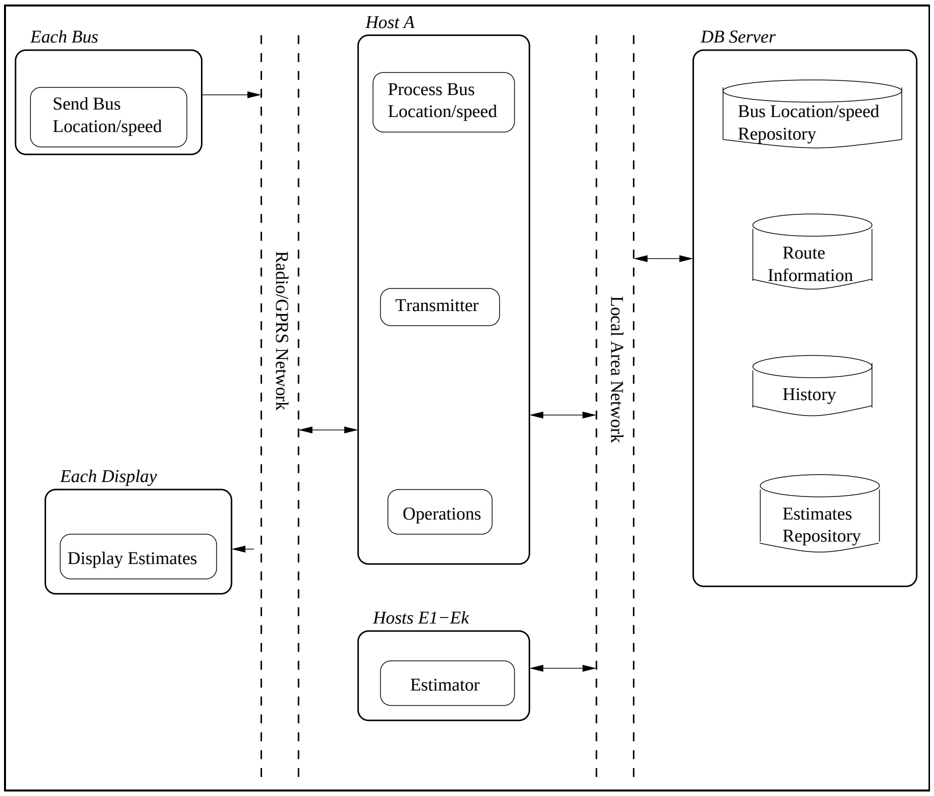

Figure 4: “Deployment” Structure — Few(er) processors

increase computational efficiency There are several places where use of an inefficient algorithm may result in the system not meeting the performance requirements, such as the estimation algorithm, or the algorithm used to construct the messages send around the system.

reduce computational overhead There are choices to be made as to whether to co-locate a passive process with an active process. Doing so will reduce the overhead of the remote communication that would be required if they aren’t co-located.

manage event rate There is a choice to be made about the frequency with which the bus subsystem sends out its location and speed.

control frequency of sampling Even if the bus sends out its location and speed at the stated rate, there is a choice to be made about how many of the messages are actually processed.

introduce concurrency There are at least as many processes as there are buses. There are several ways in which these processes can be allocated to processors. Some aspects of the performance can be improved by using more processors.

increase available resources Another choice that can be made is that faster processors can be used, rather than more processors (or both). There is also the possibility of using separate communication links for different parts of the system.

maintain multiple copies of either data or computations Multiple copies of the **Estimator** process are used, one per bus. It is also possible to duplicate the route information or history information.

Justification

The **Transmitter** process is initiated every 15 seconds by a clock in the **Operations** process. In order to determine whether the architecture meets the stated performance requirements, there are two questions of the **Transmitter** process’ behaviour that need to be answered:

1. Does the process send estimates for all buses currently on the road before the next clock tick?

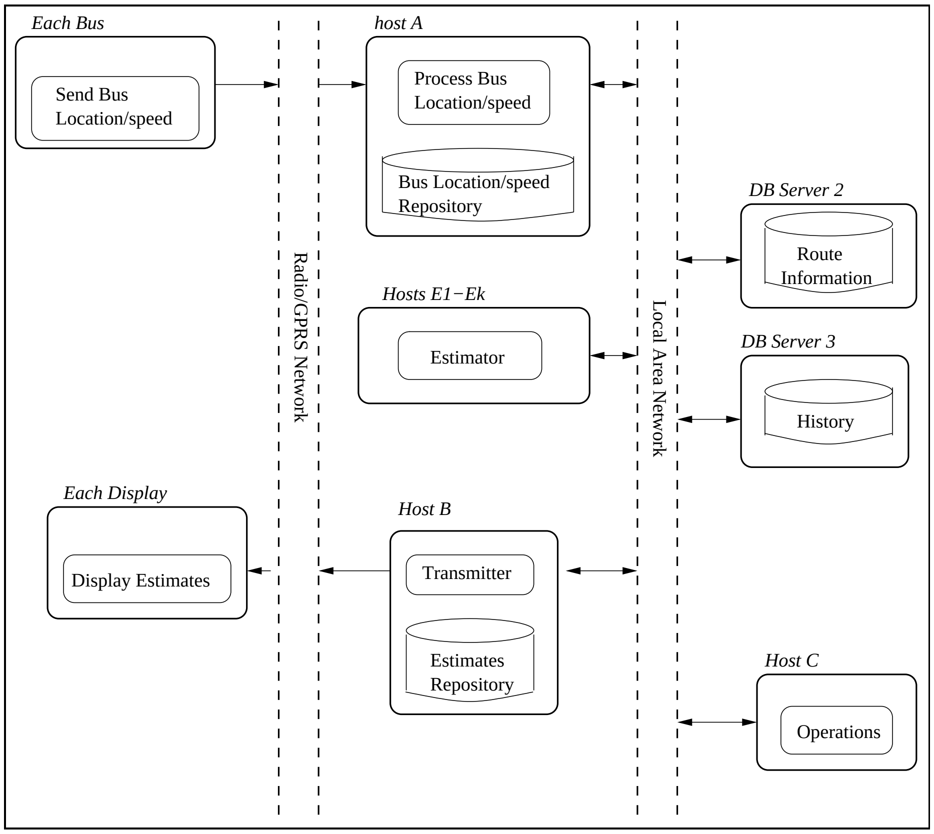

Figure 5: “Deployment” Structure — Combining active and passive processes

2. Are the estimates sent within the required accuracy?

The **Transmitter** process has to do two things: get the list of estimates from the **Estimates Repository**, and then send it all out. We have been told that there are at most 1000 buses on the road at any time, and that for each bus there are at most 5 major bus stops on its route and so at most 5 displays to be updated. In other words, there are at most 5 estimates per bus that need to be extracted from the repository and sent out, or 5000 overall.

Extracting 5000 objects from the **Estimates Repository** process should take milliseconds. If sending the result across the network takes too long (although at local area network speeds this seems unlikely), then **reduce computational overhead** by locating the **Estimates Repository** process on the same processor as the **Transmitter**.

Assuming that what is sent for each bus is its route number (at most 4 characters) and its estimated arrival time (assume 24 hour clock, and each estimate is to the nearest minute, so 4 characters), then we can be sure we only need to send out 10 bytes per estimate,. This means that, overall, we need to send out 50000 bytes, or 50KB. We have been told that the transmission speed is at least 60KB/second, so we can conclude all the information can be sent out in 1 second.

However, there is some overhead involved in grouping the estimates for each display, and then packaging up the messages to be sent to each display. However this is all done inside the process, and since that process can have a dedicated processor if necessary, this *should* be doable within 1 second, giving an overall time of 2 seconds.

In fact, constructing the strings from the information that makes up the estimate will take the longest of all the tasks the **Transmitter** has to do. If the implementation is in Java, then just using the String concatenation method may take too long, but careful implementation (i.e., **increase computational efficiency**) will deal with this problem.

From this it seems believable that the **Transmitter** process can send all the estimates out well within the 15 seconds it has. However, there’s not point being able to send out information if the information isn’t accurate enough. The next question is, how up to date can the estimates managed by the **Estimates Repository** be? This can be determined by how up to date the bus location and speed information is, and how long the estimation algorithm takes to run.

If there are 1000 buses on the road, and the messages containing their current location and speed takes at most 16 bytes (4 bytes for id, 3 bytes for speed, 9 bytes for location), then that’s 16Kb being sent at the most. The distance the messages are transmitted is relatively small (compared to the speed of light), however they also have to be sent from the receiver via the Internet to the central control system. Nevertheless, it's believable that all the data can arrive at the **Process Bus Location/speed** process within a second of it being sent. This suggests that a single **Process Bus Location/speed** process can cope with all of the buses sending their location and speed every second. If it cannot, then we can either run the process on a faster computer (**increase available resources**) or run more than one **Process Bus Location/speed** process each on its own computer (**introduce concurrency**). Note that the second option will mean that network communication will be needed to get to the **Bus Location/speed Repository** process.

If the **Bus Location/speed Repository** is indexed by bus identifier and uses an efficient data structure for searching on the identifier, such as a hash table (i.e., **choose a more efficient algorithm**), then updating the repository with the new locations and speeds ought to take milliseconds, unless it has to go across a network, and even then it should not take more than 1 second at local area network speeds.

Each **Estimator** process will request the current location and speed for the bus that it is responsible for, which will require a few bytes to go across the network. Once it's computed the new estimate, that will be sent to the **Estimates Repository**. The time taken to receive and send the information again should be quite small, and certainly less than 1 second.

The **Estimator** process has to also get route and history information relevant to the bus. The route information is needed to determine where the bus is on the route, and so which bus stops that estimates need to be computed for and also to determine the bus' scheduled arrival time. A reasonable organisation of the data (**increase computational efficiency**) should allow this to be done within milliseconds. The history information needed is the time taken by the bus from its current location to each of the remaining major bus stops on the route in the past. This could be a lot of data (e.g., times for each display for each day going back 2 years — possibly 3-4000 values), however most of the time the time taken will be about the same, or there will be only a few different values (e.g., two different times for when schools are on holiday when when they're not), so we should be able to pre-process this data off-line to reduce the amount needed (**increase computational efficiency** again). Let us assume that getting all of this information takes 1 second.

From this, we can conclude that the time taken from when the location and speed is determined by the GPS, and an estimate is received by the **Estimates Repository** process, is 4 seconds plus however long the estimation takes.

The actual time taken by the estimation algorithm will depend on the speed of the machine the **Estimator** process is running on, as well as efficiency of the algorithm. We can change the hardware (**increase available resources** and **introduce concurrency**), and may be able to change the algorithm (**increase computational efficiency**). If we *assume* that the estimation algorithm takes not more than 10 seconds per bus, then that means we can update the estimate every 15 seconds.

There is a question of network contention between the (possibly 1000) **Estimator** processes and other processes they communicate with. However, each process will receive at most 16 bytes from the **Bus Location/speed Repository** process, perhaps 1Kb each from the **Route Information** and **History** processes, and send at most 16 bytes to the **Estimates Repository** process. Overall, that's about 2 MB, which will be spread out over the 15 seconds mentioned above. This is not going create problems at local area network speeds (100Mbps).

Note that since we're only using the current location and speed information every 15 seconds, we can afford to reduce either the send rate from the buses (**manage event rate**) or ignore many of the messages coming in (**control frequency of sampling**).

If the network becomes a problem, we can have separate networks (**increase available resources**) and have multiple copies of some of the information, such as the route information and history (**maintain multiple copies of either data or computations**).

The accuracy requirements are that the estimated time be within 2 minutes of the actual time when the bus is within 1 kilometre of the bus stop. If a bus is travelling on average 30 km/h, then in 2 minutes it can travel 1 kilometre. If we are updating the estimates every 15 seconds, then the estimates should be sufficiently accurate, assuming the estimation algorithm is good enough.