A UNIX† Task Broker

Andrew McRae

Megadata Pty Ltd.
andrew@megadata.mega.oz.au

This abstract describes a UNIX Task Broker, an application which provides redundant processing configurations using multiple hetrogeneous hosts connected via a network.

Firstly some background is provided to explain the context in which the Task Broker was developed; then the overall functionality and environment is discussed. Some closer detail is given of the various components that make up the system, and some real world results of a delivered system are reported. Finally some broader applications are discussed, such as how the Task Broker can be used to provide a generic solution to many problems found in a distributed environment.

1. Introduction.

The growth and development of the distributed computing environment has brought with it mixed blessings; on one hand cheap and affordable computing power has meant a major increase in the amount of CPU performance able to be applied to data processing; on the other hand it is often difficult (especially with a hetrogeneous distributed environment) to rationally apply the performance in a manner which guarantees high user availability and robustness, yet providing application invisibility and best use of resources.

To solve the general problem of high availability and robustness in the UNIX environment, Megadata has developed a distributed Task Broker system, comprising an application task manager and a services directory manager. The combination of these modules allow a group of diverse networked Unix systems to run a set of application services providing mutual backup in the event of host or network failure (for high reliability multiple physical networks remove any single point of failure). A Services Directory protocol allows dynamic discovery of available services for client applications, and automatic re-homing to a new server in the event of host or software failure.

One of the most interesting aspects of this system is the fact that systems of entirely different architecture (and systems from different vendors) may form a highly reliable network that is expandable and robust, where applications will ‘fail-over’ from one system to the other. This gives the customer independance from any one vendor’s system, demonstrating that Open Systems are a reality in the real time world.

2. Background.

Traditional fault tolerant systems rely on proprietary hardware and software to ensure a guaranteed level of availability. Availability is measured as a percentage of up-time to down-time; typical real time systems must meet or exceed 99.9% availability under all operating conditions, which is equal to less than 9 hours down-time in a year. A goal is to achieve another order of magnitude of availability (99.99% - less than an hour of down-time per year). As an example, one system Megadata has had operational in the field since 1983 has experienced an accumulative total of 16 minutes downtime.

To achieve this level of availability, real time systems usually relied on duplication (or triplication) of critical components such as CPUs, discs and other peripherals, all of which operated on proprietary bus interconnection, and relied on specialised inter-processor communication to detect host hardware or

---

† UNIX is a trademark of Bell Laboratories.

software failure (Figure 1). Because of the duplicated bus, generally only one CPU was online performing all necessary functions, with the other running in a monitoring standby mode.

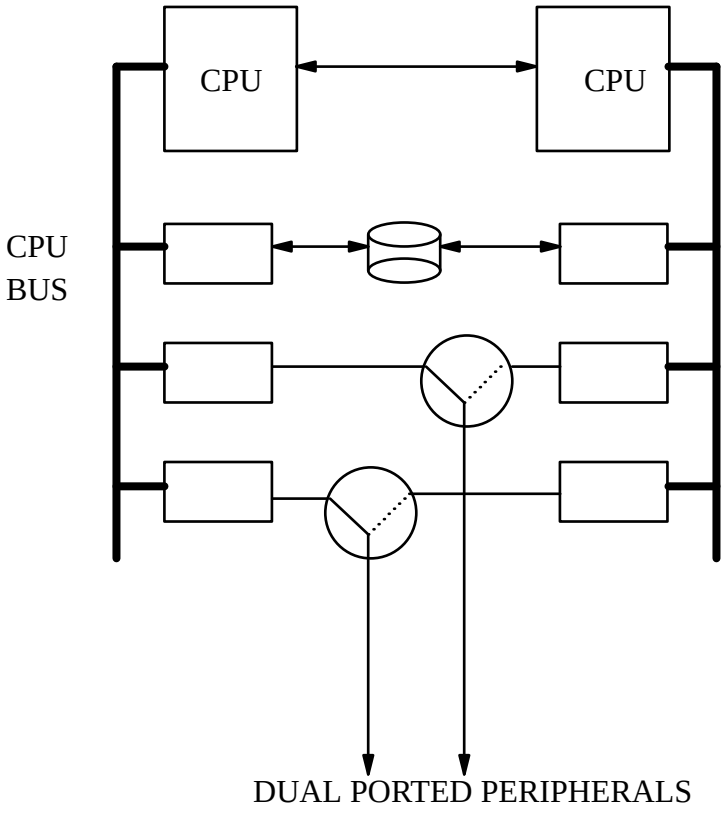


Figure 1

One side effect of this architecture is expense. It costs a lot of money both in actual hardware due to duplicated subsystems and peripherals, and in the design, support and commissioning of the specialised hardware and software. Another cost is the inability to keep up with new technology, as generally the proprietary nature of the products require a major investment in time to develop and prove. This is a valid point in favour of Open Systems in general, though especially relevant in the real time field which is traditionally slow to accept new technology.

Another major factor is that the implementor is at the mercy of the vendor supplying the major components, and cannot take advantage of any price/performance benefits offered by other vendors.

When Megadata started developing real time applications under UNIX instead of a proprietary architecture, it heralded a radical change in the approach to developing high availability, redundant real time systems. One point to highlight at this juncture is that Megadata real time systems are not aimed at high speed laboratory data acquisition, but at high availability supervisory and control systems, in which the time scale is on the order of tens or hundreds of milliseconds, not microseconds. Specialised data acquisition hardware performs the *real* work of obtaining the telemetry data.

The problem that faced us was how to achieve high availability and fault tolerance using standard ‘Open Systems’ i.e. Unix workstations without specialised hardware.

After some evolution, the following diagram shows a new UNIX based architecture replacing the older style architecture (figure 2).

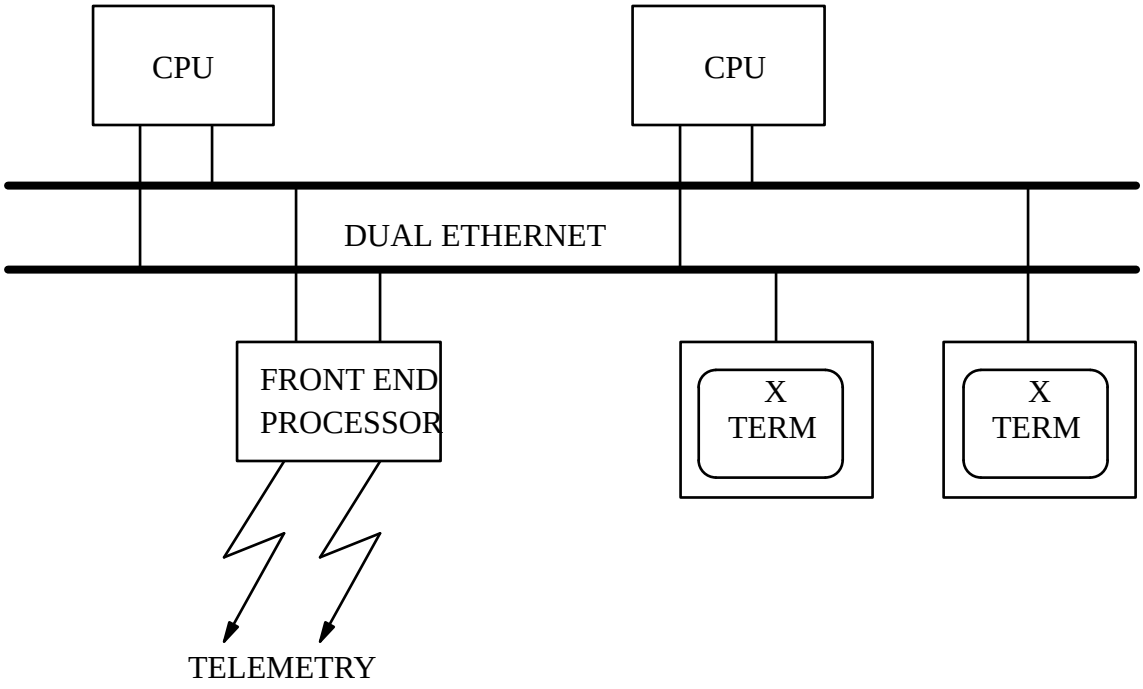


Figure 2

The new hardware architecture is orientated around a dual Ethernet LAN (for even greater levels of fault tolerance more LAN sub-nets or splitting segments with bridges may be considered), with two or more hosts acting as central processing nodes, and peripherals distributed along one or both LANs.

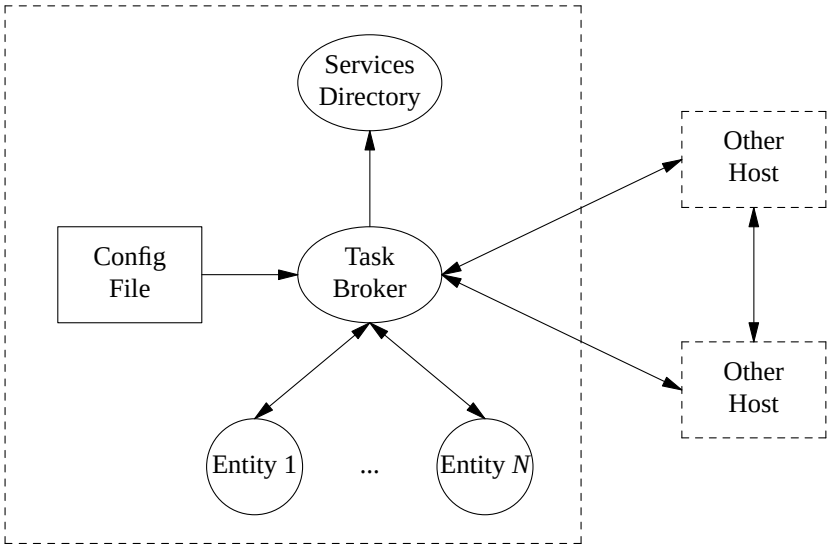
A critical component of this grand scheme is the software equivalent of the specialised hardware and software that controlled the software and host failure handling, which has evolved into a general software solution to maintaining high availability of services on multiple hosts on a network.

3. Overall Design.

The UNIX Task Broker basically provides host independent *service migration* (as opposed to *process* migration). True process migration operates by moving a complete process image (including file descriptors, memory image, process state etc.) transparently between different CPUs, either to load balance or to cater for processor failure. Very few systems provide this level of redundancy, and it requires a large degree of operating system support, high network bandwidth and network protocol support to achieve it. Standard UNIX (if such a beast ever really exists), does not provide any support for process migration.

Service migration, on the other hand, is the ability for clients to dynamically discover hosts which are providing required services, and then if some failure causes the host to stop providing the service, then another host will ‘pick up’ the service and initiate action whereby the client(s) of the failed host can transfer to the new host. In other words, the service *migrates* to a new host. Two elements work in conjunction to provide this action. The **Task Broker** performs the role of detecting host failures and ensuring that at least one host is providing service to the clients; the **Services Directory** provides a dynamic database of services and hosts that clients query to discover a host that can provide a particular service. The Task Broker co-operates with the Services Directory by informing it when hosts or services are not available.

The Task Broker appears thus:



Each host that is involved in the system runs an instance of the Task Broker. Each host also runs a Services Directory server, which operates on a well known UDP port number.

The tasks that the Task Broker controls are known as *entities*, where each entity is one or more UNIX processes. For an entity to be operational (or *online*), all processes in the entity group must be running. If any process dies, or is otherwise unrunnable, then the whole entity is considered *offline*.

Multiple instances of the Task Broker may run on the same host, and each is assigned a separate network port to communicate with other Task Brokers. Thus it is possible to have completely separate and distinct *systems* running on a group of hosts. Each system is uniquely identified via an defined name, known as a *System Identification*, or just System ID. One only Services Directory server runs on each host, thus when a client wishes to connect to a service the identification must include a service name and a System ID.

A configuration file is used by the Task Broker to describe the hosts upon which entities are to be run, and each entity is defined (specifying the programs to run, arguments, home directory etc.). Each Task Broker in the same system on different hosts shares the same configuration file (making it easy to maintain).

4. Host Failure.

When several Task Brokers are operational in a system, they continually monitor each other by broadcasting to each other their current state (including the list of entities that are currently running on this host). The broadcast interval is settable, and determines how long before host failures are detected. If two or more broadcasts are missed from one host than the other hosts poll the (possibly failed) host, and if no reply is obtained then the host is considered failed. The entities that the host was running are re-arbitrated amongst the remaining hosts.

Dual physical networks are provided to remove the possibility of some kind of network problem, and the Task Broker will always use both physical paths to communicate to remote Task Brokers.

Using a broadcast interval of 1 second (the lowest allowed) host failure can be detected within 5 seconds, and depending on the time it takes for entities to initialize, clients can be informed of the new server and be re-homed to the host providing the service within 8 to 10 seconds of the failure of the original hosts. A broadcast interval of one second does not generate excessive network traffic, nor does it load the hosts excessively (setting slower rates will proportionally relax the timing constraints).

5. Task Arbitration.

When Task Brokers are started on hosts within the same system, the configuration file will specify which hosts are allowed to run the list of entities. Two types of host lists are attached to each entity. A *preferred* host list specifies a list of hosts for running a particular entity. The order of hosts is important, as they are assumed to be in priority order. Lower priority hosts will only start the entities if (and only if) the higher priority hosts are deemed to be inoperative. If a lower priority host is running an entity, and a host that is marked (for this entity) as being a more preferred host becomes available, then the lower priority host will yield up the entity to the preferred host.

Another host list may be specified either instead of, or extra to, the preferred list. This list places no priority onto the hosts, so arbitration becomes somewhat more random.

If an entity has no clear priority indicating which host to run on, the Task Broker on each host arbitrates by sending a *claim entity* message to all the other hosts that the entity can operate on. Attached is a psuedo-random number, and if two Task Brokers collide when arbitrating the random number is used as a final judge.

A number of rules govern the arbitration process, ensuring that the same entity cannot be accidently started on different hosts at the same time. Basically every host that starts an entity **must** have obtained a go-ahead from every host capable of running this entity.

6. Services Directory.

The Task Broker program ensures that selected services are always available on a host somewhere on the network, but how do clients discover which host is providing the service? Beyond discovering the host, it is possible that multiple instances of the service is running (one for each distinct system operating on the LAN), and the client might have to be told which network port number to use etc. The Services Directory combines with the Task Broker to solve this problem. A directory is maintained on each host of the available *services* that are operating on this host, and clients may discover which host is providing a service via a Services Directory Protocol to the well known Services Directory UDP port. This is somewhat reminiscent of the Sun RPC/Portmapper situation, where RPC services register with a central portmapper program; the difference is that the Services Directory is quite dynamic, depending on whether particular entities are operational or not.

Each service is identified via a name, and services may be associated with an entity, so that the Task Broker can inform the Services Directory server on each host whether the entity is running or not (and hence whether the service is available). Arbitrary options and option data may be attached to services so that clients can obtain not only the host address providing the service, but can be given other information, such as the network port number, initial state, and other application data.

The steps a client would perform to connect to a named service (for example the service may be called **LOGIN**) would be:

1) Broadcast a service request to the well known Services Directory port, containing the service name.
2) Every host that is able to provide the service replies with a service reply that specifies the IP address of the host, as well as any other options or data associated with the service.
3) The client (presumably) will receive one or more replies, and depending on the client may accept the first one received, or may delay and select the host depending on the options and data in the reply. If no reply is received the client retries the broadcast.
4) Once connected to the service, the client may listen for a service reset for that service on a well known port. When a service reset is received, this indicates that the host is no longer able to provide the service (possibly due to some hardware or software failure). The client can then re-discover which host is now providing the service via the same process as above.

The service reset packet is usually broadcast when a server becomes operational on a new host, as a result of another server becoming unavailable on a failed host. The reset will trigger all existing clients to transfer to the new host. For UNIX clients that cannot listen on the well-known port, a local connection to the Services Directory server will allow the server to intercept the reset and inform the client when a reset packet is received.

7. Broader Applications.

Whilst the Task Broker and Services Directory were originally conceived and implemented for the real time area, a much broader scope of applications can benefit from the dynamic nature of service migration and discovery.

One simple example is load balancing, where users have multiple machines on a network available for various applications; when logging on, the users can run a telnet front end which performs a service request and discovers the machine which has least load or fewest users, and then running a real telnet to that host. A similar scheme can exist for xterms and other interactive shells. Building knowledge of the Services Directory protocol into a terminal server would be trivial.

Printer services are a natural application, where the location of a printer (or a generic printer type) and its host server is dynamically determined - extra factors such as current queue length or job size or urgency may be taken into account.

Makes may be enhanced by running compilations on lightly loaded machines. Enforcement of policy is useful, where some machines may be dedicated to certain applications, but when those hosts are down others may take on the extra load.

File servers may benefit by enabling client hosts to discover which machines have certain file systems available, and to automatically remount the file systems elsewhere in the event of file server failure (whilst automounter-type daemons perform this role already, they are limited and are often not dynamically reconfigurable).

The general concept of service migration and dynamic discovery of services work best in an environment where multiple hosts provide a range of services to clients, any of which may operate across a network. One of the more impressive demonstrations occurred when a Sun workstation and a HP workstation were providing redundancy to each other, and the client programs were able to transparently switch from one to the other upon host failure, and back again.