6. Prototype Evaluation

Chapter 6

Prototype Evaluation

"The mark of a truly civilised human being is the ability to read a column of numbers and then weep."

Bertrand Russell

Evaluating a system implementation can be undertaken at two levels: *component* and *system*. A component analysis examines each system component in an isolated manner whilst a system analysis is holistic and operates at a higher-level, considering more functional problems. Indeed there is a fine balance to be struck between being too specific which produces results that do not mean anything useful, and being so general that there is no content to the results. The component level provides useful information that can aid development and testing but suffers from a lack of relevance when a system task is considered. At the system level the whole system is asked to perform some useful task and evaluation of its performance can be used to judge its overall effectiveness.

These methods are not mutually exclusive, in fact understanding system performance is difficult if the effects that the individual system components have are not fully understood. However, a component's behaviour will often change when used in conjunction with other components within a system, e.g. its performance may be reduced when it has to bid for CPU time with other processes. This chapter, therefore, deals with the system as a whole (an approach advocated by Checkland, 1994) but with a detailed look at the two major components of most (if not all) system processes: the UML interpreter and the PML.

6.1 System Analysis

Ideally one would like to compare the performance of this prototype with that of other solutions for distributed VE systems. However, the only evaluation of a VE system that the author has found is for AVIARY (section 2.3.8), with the exception of a predictive performance chart for DIS (Figure 2.3). Even if figures were readily available, the problems that must be faced when comparing systems are similar to those encountered when comparing CIGs. Each manufacturer presents a list of figures which detail the CIG's performance of certain tasks, e.g. rendering a 10 by 10 grid of polygons, in relatively useful units, e.g. polygons per second. Unfortunately, information essential for comparison of the CIG's results

with another CIG is often not presented, e.g. were the polygons lit? Clipped? Textured? With which texturing technique? etc.

The obvious course of action would be to derive a set of benchmarks that may be used to provide a fair basis for comparison of systems. But even this has problems, for example some CIGs are optimised for triangles whilst others can handle polygons with any number of vertices. Undoubtedly, any test using triangles will give any CIG optimised for this type a better rating than the other CIGs. Conversely, a benchmark that tested polygon throughput with varying numbers of vertices cannot be run on a triangle-only system without extra application processing to split the polygons into triangles, thus defeating the objective. There are many other examples of architectural differences that confound comparison.

The architectures of distributed VE systems are even more diverse than that of CIGs and presents a challenge when designing benchmarks. In the same way that a geometrical model can produce different performance ratings on different CIG's, VE system performance is very application specific. This may be a reason why figures are not available for existing systems - even the evaluation of AVIARY is based around an Air Traffic Control application. No attempt will be made in this chapter to derive a set of useful benchmarks since this is a subject suitable for a thesis in itself, a more basic approach will be used instead.

This thesis has already established that the user is the final judge of the system's effectiveness and that certain criteria must be met to provide a usable interface (section 3.3). Although this prototype was not built to test these measures, it is possible to extract the most important feature of any such system which is the ability to progress the simulation as fast as possible. A suitable metric is simulation steps per second and is used as the absolute measure of this prototype's performance.

6.2 Testing Methodology

All of the benchmarks used in this chapter were run under similar conditions. Normal operating system processes were reduced to a working minimum in order to maximise the available memory and minimise interference with the USS processes. No users were permitted access to the machines during testing and normal Internet services were suspended. Disk accesses only occurred at the beginning of a test and at the end when results were logged. Even then, only local storage was used, which was especially important in the case of the SGI where normal user directories are held remotely and accessed using the Network Filing System (NFS). This fact combined with the presence of virtual memory can drastically affect performance.

This section documents the relevant characteristics of the machines used to test the prototype and highlights a number of issues that affected system performance.

6.2.1 CPU Performance

Table 6.1 shows the relative performance of several Intel CPUs present in IBM PCs and the MIPS processor used in SGI's RealityStation. The performance ratings are, of course, dependent upon the efficiency of the compiler and its ability to generate optimised code. The Watcom C++ compiler was used on the Intel-based platforms whilst GNU C++ (G++) was

used on the SGI machine. The native C++ compiler was not used because it did not support exceptions but unfortunately the GNU compiler had a number of faults that presented problems. Firstly, the code optimiser could not be invoked if the source code used exceptions, subsequently the SGI's performance was severely undermined. The figures shown inside the brackets are those of the native C++ compiler with optimisation and those outside the brackets represent the results obtained using the GNU compiler without optimisation[1]. Secondly, the implementation of C++ templates is less than efficient with the current release of G++ and requires the instantiation of each template within each and every module it is used (GNU, 1995). Consequently, the executable sizes produced are much larger than necessary which in turn has implications for the amount of paging required during execution.

	Gateway **i486** **50 MHz**	**Server** **i486** **66 MHz**	**Pentium** **Pentium** **90 MHz**	**Reality** **MIPS 4400** **200 MHz**
Integer	0.256645	0.330643	0.849287	0.653 (1.537)
Floating-point	0.173911	0.211827	0.881350	0.517 (1.772)
Memory (available / total)	14 / 20 Mb	10 / 16 Mb	17 / 24 Mb	128 Mb + virtual memory
Bus	16 bit (ISA)	16 bit (ISA)	32/16 bit (PCI/ISA)	256 bit
Bus Speed	50 MHz	33 MHz	30 MHz	47.6 MHz
Bus Bandwidth	95 Mbytes/sec	63 Mbytes/sec	114/57 Mbytes/sec	1.42 Gbytes/sec
Disk	1 Gb	1 Gb	750 Mb	2 Gb

The CPU speeds were obtained using BYTE Magazine's BYTEmark benchmark program. A rating of 1.0 is equivalent to a DELL Pentium 90 MHz PC running DOS. The figures given include the machine's multi-tasking operating system overheads. Figures in brackets represent the native compiler's performance on the SGI.
ISA - Industry Standard Architecture
PCI - Peripheral Connect Interface

Table 6.1 Resource ratings for each test platform.

6.2.1.1 QNX

The total memory available on each platform running QNX is shown in Table 6.1 as well as the actual amount that may be used by non-system software. Since QNX does not provide any virtual memory this limits the number of system processes that may run at one time. The

[1] The benchmark code did not contain exceptions and thus could be optimised, resulting in performance only slightly worse than that produced by the native compiler. However, these results would not be indicative of the prototype's performance and hence the unoptimised figures are given.

absolute maximum number of executables running simultaneously is 250 which allows for a maximum of 50 virtual circuits[2] (QNX, 1995).

Each of the three QNX machines (Pentium, Server and Gateway) are interconnected by a private Ethernet LAN using the same make of Ethernet card and the same Industry Standard Architecture (ISA) bus. Gateway has a second interface card installed which is connected to the university's backbone network.

6.2.1.1.1 Scheduling

There are three different scheduling methods that any given process may be assigned to under QNX: *First In First Out* (FIFO), *round-robin* and *adaptive*. When using FIFO scheduling a process executes until either it voluntarily relinquishes control (blocks) or is preempted by a higher-priority process. FIFO is only of real use to ensure mutual exclusion when two processes are sharing a resource. Round-robin is like FIFO except that each process may also stop executing if it reaches the end of its timeslice (100 ms). Adaptive scheduling uses decaying priorities for those processes that consume their timeslice and priority boosts for those processes that are starved of CPU for one second or more.

The last scheduling policy is commonly used in systems where interactive and compute-intensive processes share the same machine, however it does make performance evaluation of a network of interacting processes difficult. All processes within the USS application were therefore placed in a round-robin scheduler at the same priority. This causes considerable starvation of the normal interactive processes (using the adaptive scheduler) but not to USS processes such as the Console.

6.2.1.2 IRIX

The limits imposed by IRIX on the number of executables, etc., were not reached by the prototype system and therefore did not interfere with the system testing. There were, however, two other issues which presented problems.

6.2.1.2.1 Scheduling

IRIX also supports different scheduling methods: real-time, deadline, timesharing, gang batch and batch. Normal interactive processes run in the timesharing queue while the deadline scheduler enables time constraints to be applied to a process - although its effectiveness is uncertain when invoked on a single processor system. Processes assigned to the real-time queue are guaranteed better performance than those in the timesharing and batch queues. Unfortunately, unlike QNX, only the super-user may promote processes to queues above the timesharing level. Due to present departmental policy, access to the test platform at this level was not granted to the author and therefore all USS processes were subject to adaptive scheduling in the timesharing queue.

---

[2] More virtual circuits may be supported by reducing the number of executables. There will be no such limits in the next major release of the operating system (v4.3).

6.2.1.2.2 Virtual Memory

Performance can also be compromised through the paging to and from disk that is undertaken when using virtual memory. Ideally all of each process' code and data would remain in memory, as with QNX. This is possible under IRIX but super-user access is again required and therefore all results obtained under IRIX are confounded by irregular and uncontrollable paging activity.

6.2.2 Computation and Communication

The main emphasis on the resources consumed by the prototype has been split between computation and communications. Sending/receiving messages requires CPU and therefore any computation rating is affected by communications. This relationship is examined when analysing the PML and its results can be used to aid estimation of specific service overheads, e.g. registering interest in a particular UML component. The other side of the equation in this example is the time it takes to manipulate the interpreter's data structure. Such information is provided by the section on UML which also deals with the resource that has, to date, been overlooked by system evaluations: memory.

6.2.3 Memory

Table 6.2 gives a breakdown of the sizes of each USS process in terms of code size, initialised and uninitialised data, and total executable size. A list of the required USS libraries associated with each process is also given in the table. Under IRIX the majority of an executable's size comes from other general-purpose libraries provided with the compiler, e.g. system call library, maths library, C++ iostreams library, etc. These libraries are much smaller under QNX, for example the total amount of USS code used in the mailer is 48,894 + 1,966 + 4,775 = 55,635 bytes, meaning that the system libraries account for 64,069 bytes. This is a worst case scenario since the amount of space used by these libraries will remain roughly the same for the larger executables. The figures for the data given above do not include the memory that the process may allocate during execution for dynamic data structures, etc. The large difference in IRIX and QNX code sizes is in part due to not using code optimisation and also the different CPU instruction sets.

A simple way to reduce the amount of memory required by each process is to make use of shared libraries. Such a mechanism places commonly used routines into a special library which is loaded once into main memory. A stub library is also compiled and is linked into the executable in place of the larger original. When a function in the stub library is called, the equivalent routine in the shared library is executed. In theory, the unique overheads incurred by each USS process may be reduced substantially since most libraries are used many times, e.g. the PML library and the UML interpreter.

The implementation of shared libraries under QNX is based upon the mechanism used by UNIX System V Release 3.2 which has an explicit interface for importing and exporting data into and from the shared library (QNX, 1994). Whereas managing data and code separately is a perfectly adequate approach for C-based applications, the technique cannot be extended to the object-oriented paradigm which deals with code and data together. Specifically, problems

occur with C++ when virtual functions, static initialisers or exceptions are used. Therefore it was not possible to exploit shared libraries with the QNX implementation.

System Component	OS	Code Size†	Data Size†	Total Size†	Required Libraries	Executable Size†
UML Library	IRIX	247,984	59,904	307,888	-	-
	QNX	78,191	77,940	156,131	-	-
UCL Library	IRIX	46,768	38,768	85,536	-	-
	QNX	13,478	37,803	51,281	-	-
PML Library	IRIX	52,270	26,032	78,302	-	-
	QNX	22,385	26,509	48,894	-	-
Entity Library	IRIX	32,496	4,016	36,512	-	-
	QNX	9,601	3,144	12,745	-	-
Manager Library	IRIX	33,840	3,600	37,440	-	-
	QNX	36,830	6,838	43,668	-	-
RProfile Library	IRIX	47,776	4,806	52,582	-	-
	QNX	10,436	2,296	12,732	-	-
Message Library	IRIX	9,232	2,336	11,568	-	-
	QNX	1,822	144	1,966	-	-
Mailer	IRIX	6,416	960	7,376	PML, Message	430,984
	QNX	3,756	1,019	4,775		119,704
RM	IRIX	32,976	3,568	36,544	PML, Message,	541,576
	QNX	24,971	5,275	30,246	RProfile, UCL	163,053
UM	IRIX	265,104	27,200	292,304	PML, Message,	1,180,552
	QNX	84,916	18,044	102,960	RProfile, UCL, UML	355,173
Benchmark Manager	IRIX	8,448	1,152	9,600	PML, Message, UML,	787,336
	QNX	4,153	1,192	5,345	Manager, RProfile	269,725
Benchmark Entity	IRIX	10,656	1,424	12,080	PML, Message, UML	787,336
	QNX	4,601	1,482	6,083	RProfile, Entity	265,689

†All sizes are given in bytes.

Table 6.2 Minimum memory usage of USS components.

This is not entirely true because the C system libraries are shared which, for example, means that every mailer only needs 55,635 bytes of memory, not 119,704 bytes. With USS shared libraries this could be reduced to 4,775 bytes or lower. Similar improvements would be seen for the other processes.

IRIX does support shared libraries but because the system was not available until very late in the project their potential was not explored. It is important to note, however, that the use of shared libraries would not only reduce the amount of memory required by a USS process, but should also reduce the amount of paging under IRIX. A large commonly used shared library has a greater chance of staying in physical memory than several large executables, each with their own copies.

6.2.4 Instrumentation

All of the data in this chapter was collected by instrumenting key execution paths with timing code. A suitable number of iterations were executed for each test case, e.g. message size, and the averaged data is used in the charts. The amount of iterations and the type of instrumentation used was determined by taking clock resolution and (erratic) operating system overheads into consideration. Under QNX, the system clock has a resolution of 0.1 ms and the SGI has microsecond accuracy. For events that completed faster or close to the clock resolution, such as some of the UML interpreter operations, the total time taken to perform all iterations was measured, adjusted for loop overheads and then averaged. For longer operations, such as the simulation execution stages, each iteration was measured individually and then averaged. In all cases the impact that the profiling code had on the measurements was taken into consideration.

6.3 UML

Quantifying the resources consumed by the interpreter permits the designer to gauge the impact their simulation will have on the system. To this end a series of simple benchmarks were used to establish resource consumption on each of the test platforms. Since these tests were compute bound, the same pattern of relative platform performance is evident in each test. Therefore, quite often only the figures from one platform will be used in the graphical illustrations of the results. A full table of the results upon which this subsection is based, along with the simple UML code used, may be found in Appendix C.

6.3.1 Code Size

The one disadvantage of sending UML code between processes is that a complex description can take an appreciable amount of space. Table 6.3 shows two possible techniques for reducing the size of UML code for transmission. Compression is a method that can be applied to any kind of data, but those algorithms that work on repeating patterns, such as the Free Software Foundation's GZIP, work well with textual data. Simply applying compression to the original UML description can result in approximately a 60% reduction in size. Another technique which can be used is that of tokenisation - it is unlikely that this would be used just before transmission but during the initial interpretation. Tokenisation simply replaces the language's ASCII keywords by single-byte tokens and reduces the whitespace used to a minimum. If the tokenised form is compressed the relative effects are less because tokenisation is a simple form of compression. However, compared with just compressing the original, the code can be reduced to around 35% of its original size.

Of course, compression comes at the cost of increased computational requirements. There is a minimum code size that compression will have a beneficial effect upon and, even then, the computation time sacrificed to achieve this makes the usefulness of such an operation dubious. Apart from the initial definition sent to processes upon creation, it is predicted that most UML code sent will be quite small, e.g. function invocations, minor code redefinitions, etc. It would seem practical, therefore, to restrict the use of compression to large messages and then only with hardware support.

Filename	Original	Compressed†	Tokenised	Tokenised & Compressed†
ts.uml	5910	2279 (38.5%)	4950 (83.8%)	2141 (36.2%)
base.uml	1131	469 (41.5%)	825 (73.0%)	392 (34.7%)

†The Free Software Foundation's GZIP was used to compress the ASCII UML files.
All sizes are give in bytes; percentages represent the compressed size in relation to the original size.

Table 6.3 Effects of techniques to reduce code size.

6.3.2 Primitive Types

Table 6.4 shows how much memory each primitive type uses on the test platforms. Although this is dependent on the machine's architecture and compiler rather than the operating system, the latter classification is used for convenience in this and some subsequent tables. The boolean type is much larger than it could be but alignment on a four-byte boundary simplified the state encoding/decoding routines and thus improved performance. The difference in the memory used by a string is due to the different C++ `String` class implementations provided with each compiler.

Name	Description	Usage (bytes)	
		QNX	IRIX
`Integer`	Integral number	4	4
`Real`	Floating-point number	4	4
`Boolean`	Boolean	4	4
`String`	Character string	16 + *len*	4 + *len*

Table 6.4 Memory consumption of the four primitive UML types.

6.3.3 Component Sizes

Figure 6.1 presents some simple formulae which may be used to estimate the memory usage of a UML component, from a literal to all of the modeled universes. Table 6.5 provides some approximate sizes of each component. The basic overheads are those that are needed merely to declare the relevant component; this will include the requirements of the base class if it is a derived component. Those overheads that are dependent on the definition being interpreted, e.g. adding a property to an element, are specified on an individual basis. These figures do not represent the variable amounts of dynamic memory that may be used in the basic overheads, e.g. the storage of strings representing names, etc. Therefore the total obtained from the use of this table will always be less than the actual memory usage. In addition the values given are dependent upon the hardware architecture (section 5.3.4) and the C++ compiler used. For example, there is no standard method of implementation to handle virtual functions in derived classes. The remainder of this section presents brief textual notes on each of the main components.

6.3.3.1 Component

All UML components are derived from the one base class, `UMLComponent`. An overhead is incurred for each dependency associated with a component in addition to the actual dependency structure. The skeleton dependency provided with the UML library only holds a single flag but, as shown in section 5.6.3, the extensions added by each application must be incorporated into this figure.

```
Component  =  Basic + (number of dependents * (overhead + Dependency))
Dependency =  Basic + [size of derivatives]

Literal    =  Basic + [length of string]
Constant   =  Basic + Component +  (number of literals * (overhead * Literal))

Function   =  Basic + Component + [return type]

Element    =  Basic + Component + (number of elements * (overhead + Element)) +
                                  (number of properties * (overhead + Property)) +
                                  (number of functions * (overhead + Function)) +
                                  (number of constants * (overhead + Constant)) +
                                  (number of converters * (overhead + Converter))

Property   =  Basic + Component + (number of instances * (overhead + Instance))
Instance   =  Basic + (size of list * overhead per list entry)

Universe   =  Basic + Component + (number of elements * (overhead + Element)) +
                                  (number of properties * (overhead + Property)) +
                                  (number of functions * (overhead + Function)) +
                                  (number of constants * (overhead + Constant)) +
                                  (number of converters * (overhead + Converter))

Entity     =  Basic + Component + (number of constants * (overhead + Constant))
                                  (number of functions * (overhead + Function))

UML        =  Basic + Component + (number of universes * (overhead + Universe))
                                  (number of entities * (overhead + Entity))
```

N.B. Square brackets [] represent optional portions of a component.

Figure 6.1 Basic relationships between UML components and their memory usage.

6.3.3.2 Literal

A literal stores either an integer, a floating-point number, a boolean flag or a character string. Dynamic memory is only allocated when storing a string, the amount being dependent upon its length.

6.3.3.3 Constant

A constant may be a list in which case an overhead is present for each list element, plus the actual size of each `Literal`.

6.3.3.4 Function

The memory used by a function is substantially increased when a return type has been declared.

6.3.3.5 Element

As one of the container components, an element can use greatly varying amounts of memory. Essentially, each component contained within the element requires information to be stored about its location.

Name	Description	Usage (bytes)	
		QNX	IRIX
`Component`	Basic	48	36
	Overhead per `Dependency`	12	12
`Dependency`	Basic	4	4
`Literal`	Basic	8	6
	Overhead for a string of *len* characters	*len*	*len*
`Constant`	Basic + `Component`	76	56
`Function`	Basic + `Component`	76	68
	Optional return type	44	20
`Element`	Basic + `Component`	140	116
	Overhead per `Element`	12	12
	Overhead per `Property`	12	12
	Overhead per `Function`	12	12
	Overhead per `Constant`	12	12
	Overhead per `Converter`	12	12
`Property`	Basic + `Component`	88	64
	Overhead per `Instance`	12	12
`Instance`	Basic	24	24
	Overhead per list entry	12	12
`Universe`	Basic + `Component`	128	104
	Overhead per `Element`	12	12
	Overhead per `Property`	12	12
	Overhead per `Function`	12	12
	Overhead per `Constant`	12	12
	Overhead per `Converter`	12	12
`Entity`	Basic + `Component`	92	68
	Overhead per `Constant`	12	12
	Overhead per `Function`	12	12
`UML`	Basic + `Component`	76	64
	Overhead per `Universe`	12	12
	Overhead per `Entity`	12	12

Table 6.5 Approximate memory usage for UML components.

6.3.3.6 Property

Whilst a property declaration is only held once in memory, the bulk of the memory consumption attributed to it is used when instancing it.

6.3.3.7 Instance

An instance is a list of pointers to the actual instance data. Therefore, each list entry incurs an overhead in addition to the actual data size which can vary from 4 bytes for most primitives, to any amount for an element.

6.3.3.8 Universe

The type of overheads detailed in Figure 6.1 are the same as those for an element except that the minimum size is slightly smaller.

6.3.3.9 Entity

The overheads for an entity are relatively small currently because instruction code is not stored, only constants and functions.

6.3.3.10 UML

A single instance of the UML interpreter holds references to all of the universes defined and the entities that exist within them. Following the data structure tree from this point enables us to determine the amount of memory used by the interpreter.

6.3.3.11 Example

Table 6.6 shows a small segment of a UML data definition. Using the data for QNX presented above, it shows how much memory would be used to represent the definition's structure within the interpreter and hold a single instance of element `Triangle`. Each component within an element automatically generates a 12 byte administration overhead in addition to the structure needed to hold that component's information. When creating an instance of a property, a 24 byte administration overhead is incurred and a further 12 bytes for every entry in a list. In the case of the `coord` array, this means that 60 bytes are used to manage 12 bytes of actual instance data, whereas 60 bytes are used to manage 216 bytes of the instance data for `vertexList`.

Any instance of a property with a primitive type will have a disproportionate amount of memory used to manage the instance versus storing the instance data. The reasons for this complexity have already been discussed (section 5.5.3). Although a special arrangement might be made for properties of a primitive type, this would make the interpreter more complex and probably increase execution time.

UML Definition	Representation Size	vertexList Instance Size	coord Instance Size

ELEMENT Triangle
{
ELEMENT Vertex
{
PROPERTY coord : REAL[3];
}
PROPERTY vertexList : Vertex[3];
}
	140
	12 + 140
	12 + 88
	12 + 88
	492
		24 + (3 * (12 + **72**))
		276
			24 + (3 * (12 + 4))
			72

Table 6.6 Example of how much memory is allocated to represent a UML definition and hold its instance data under QNX.

Component	Action	Pentium (ms)	Server (ms)	Gateway (ms)	Reality (ms)
ELEMENT	Insert	0.308	1.160	1.032	0.284
	Replace	0.376	1.397	1.192	0.314
	Delete	0.313	1.117	0.946	0.244
CONSTANT	Insert	0.339	1.202	1.094	0.303
	Replace	0.381	3.911	1.202	0.318
	Delete	0.297	1.060	0.917	0.230
PROPERTY	Insert	0.342	1.217	1.097	0.301
	Replace	0.372	4.752	1.205	0.316
	Delete	0.298	1.073	0.939	0.230
FUNCTION	Insert	0.320	2.430	1.039	0.288
	Replace	0.353	3.853	1.149	0.304
	Delete	0.288	1.067	0.909	0.226
ENTITY	Insert	0.302	2.102	0.968	0.282
	Replace	0.324	3.237	1.046	0.286
	Delete	0.325	1.156	1.029	0.260
Dependency	Add	0.005	0.047	0.059	0.021
	Delete	0.019	0.184	0.076	0.039

Table 6.7 Fundamental interpreter operations timings for each test platform.

6.3.4 Interpretation

Figure 6.2 shows the relative time taken to perform the three basic interpretation operations (insert, replace and delete) for three primitive components on each of the test platforms. In the case of the element and entity components, the definitions used in the test had no contents so that the measurements would be representative of each component. The property was given an arbitrary primitive type (integer) for the same reasons. Similar measurements were performed for functions and constants but give results very close to that of the property

because the same amount of memory is currently used to represent them internally. Complete details may be found in Table 6.7.

Figure 6.3 shows how long it takes to add and remove a dependency for any component. The actual time taken to perform this operation is dependent on the number of existing dependencies on the component and its position within the dependency list. The results shown here are, therefore, the best case results.

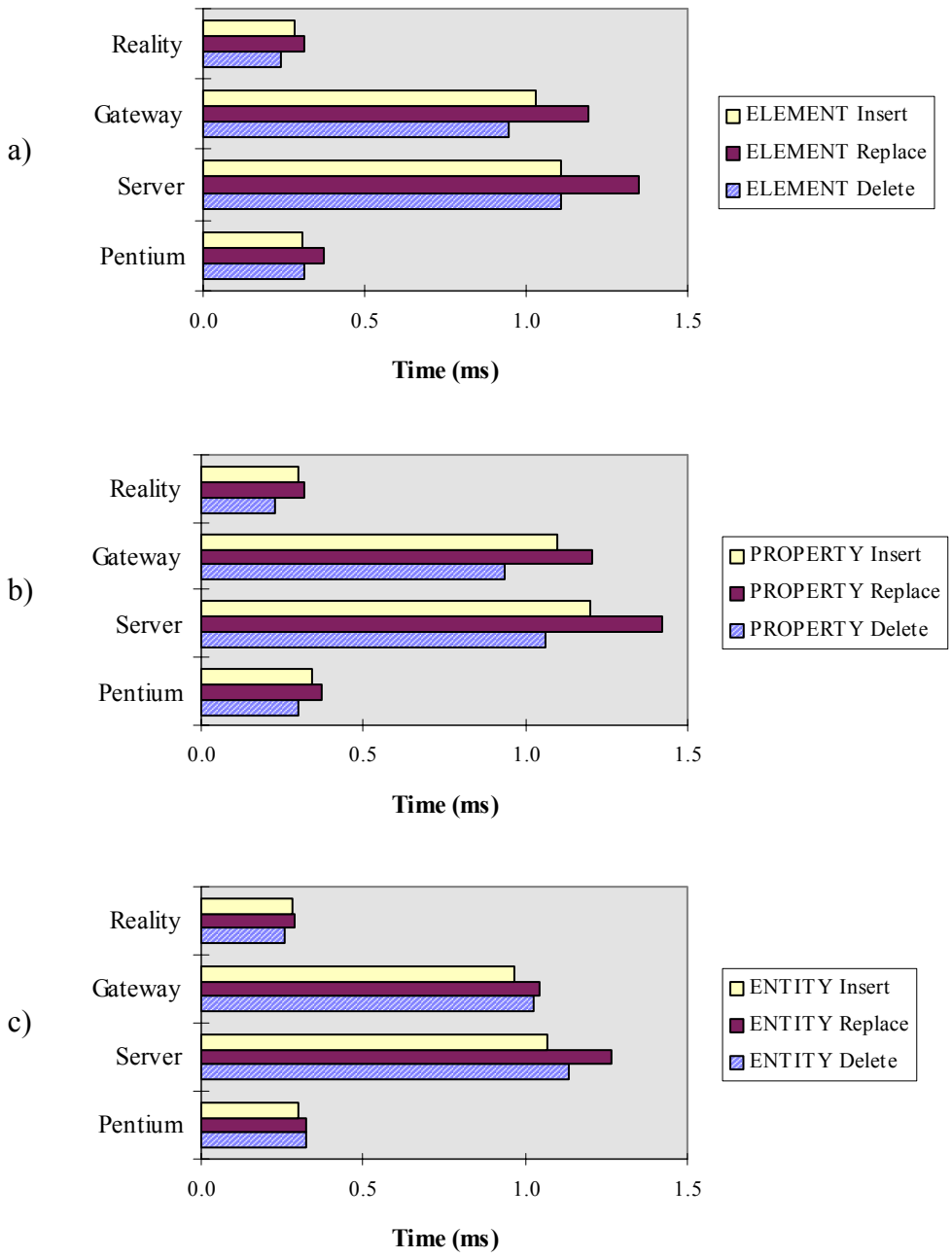


Figure 6.2 Basic interpreter overheads for three primitive types: a) Element; b) Property; c) Entity.

6.3.5 State Management

The five operations that are performed on a component's state are those for instance control: construct-destruct, and those needed for state encoding: size-pack-unpack. The duration of these compute-bound operations on an integer, real or boolean is shown in Figure 6.4 (an empty string increases the time for these actions marginally due to its slightly larger size). There is a linear relationship between state size and operation performance, and the time taken to complete any operation is extended if the component is an array.

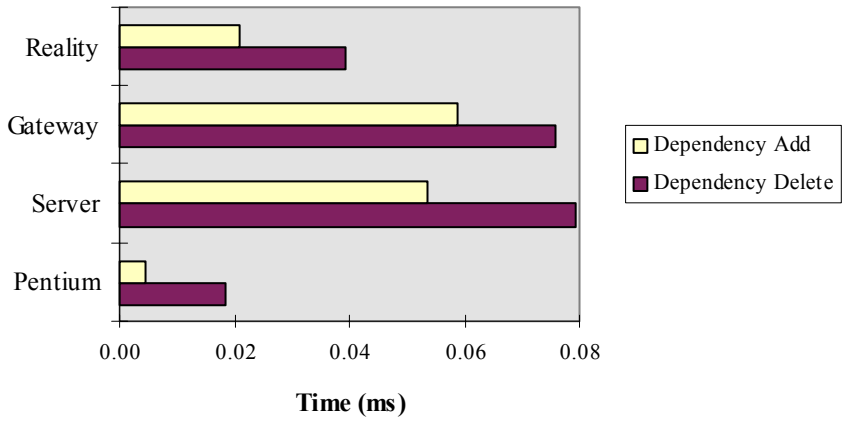


Figure 6.3 Cost of adding and removing a dependency on a UML component.

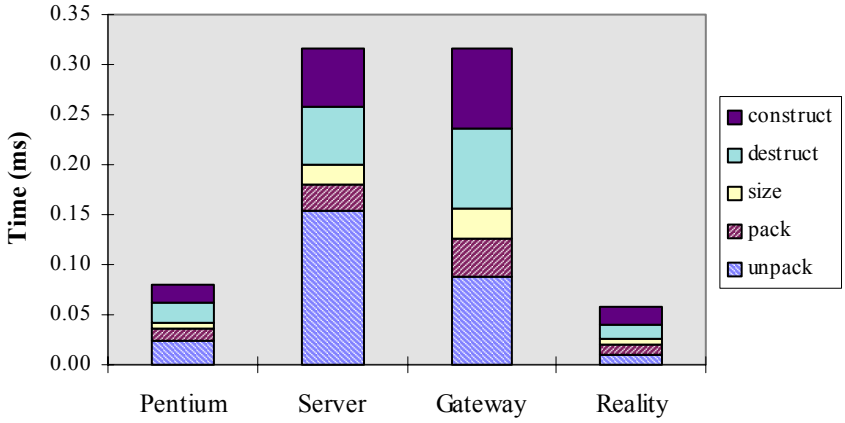


Figure 6.4 Fundamental state operations on an Integer/Real/Boolean and their cost on each platform.

To examine the impact that state size has on the performance of each operation, the length of an array of integers was varied and resultant times recorded. Figure 6.5 is a graphical representation of the results whilst Table 6.8 details the time increase of operation execution if one element is added to the array.

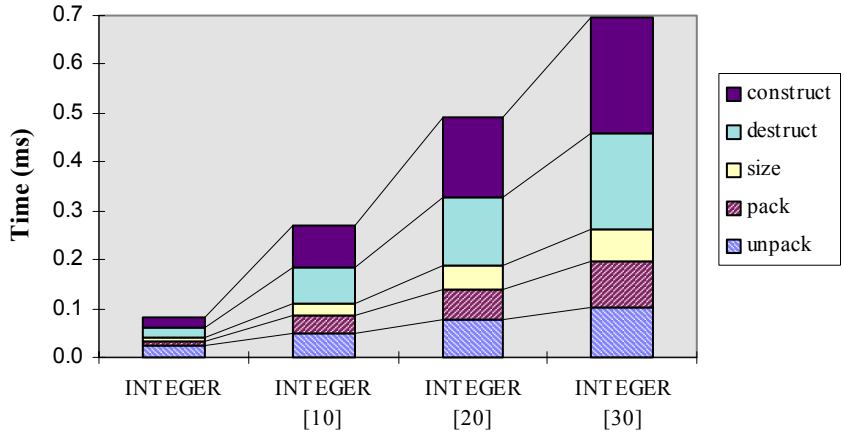


Figure 6.5 State operation costs based upon state size (Pentium).

Operation	Pentium Time (ms)	Server Time (ms)	Gateway Time (ms)	Reality Time (ms)
construct	0.008	0.024	0.035	0.006
destruct	0.006	0.018	0.027	0.005
size	0.002	0.006	0.009	0.001
pack	0.003	0.015	0.012	0.002
unpack	0.003	0.016	0.011	0.002

Table 6.8 Operation overheads per integer array element.

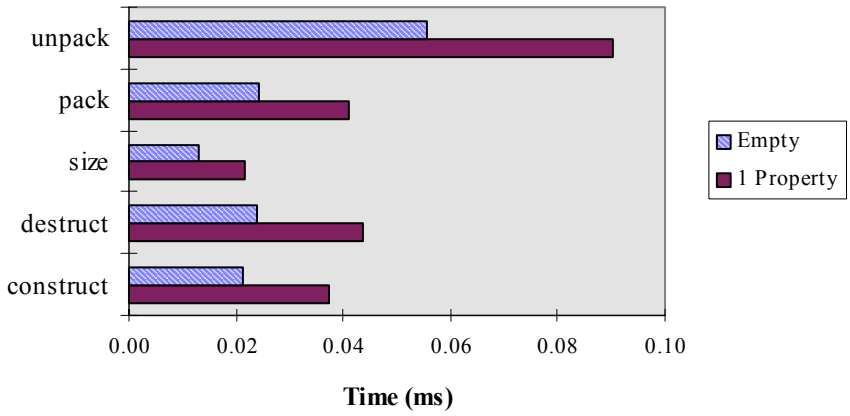


Figure 6.6 State operation overheads for an element with zero and one properties (Pentium).

The performance of the state operations on an empty element compared to that of an element with a single property (an integer) may be viewed in Figure 6.6. Unsurprisingly the difference is equal to that of a single property (Figure 6.4), therefore the computational cost of managing an element may be calculated by totalling the costs of the individual properties contained therein, added to the basic element overhead.

Similarly, nested elements produce predictable results (Figure 6.7), each level comes at the price of a single element's overheads.

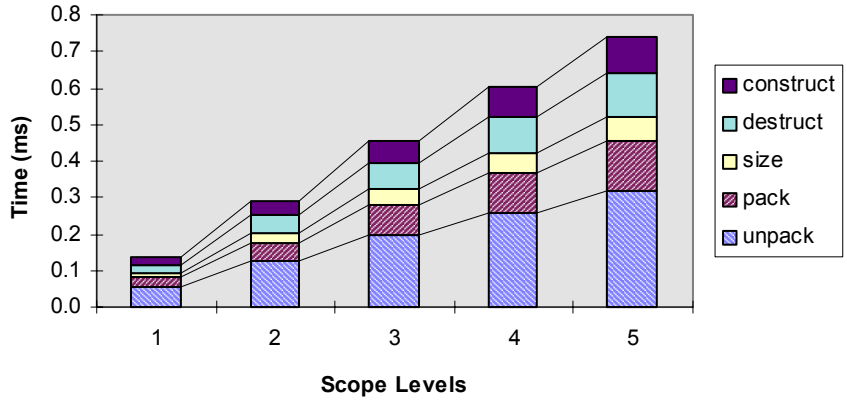


Figure 6.7 State operations on elements nested 1 to 5 levels deep (Pentium).

The cost of reinterpreting any part of a UML definition may be estimated by determining the differences between the current and new definition. Those parts that are now obsolete must have their state destructed and then the relevant portion of the data structure removed. New component definitions are added in the normal way whilst those components that are redefined require partial (or complete) state destruction, re-interpretation and re-construction.

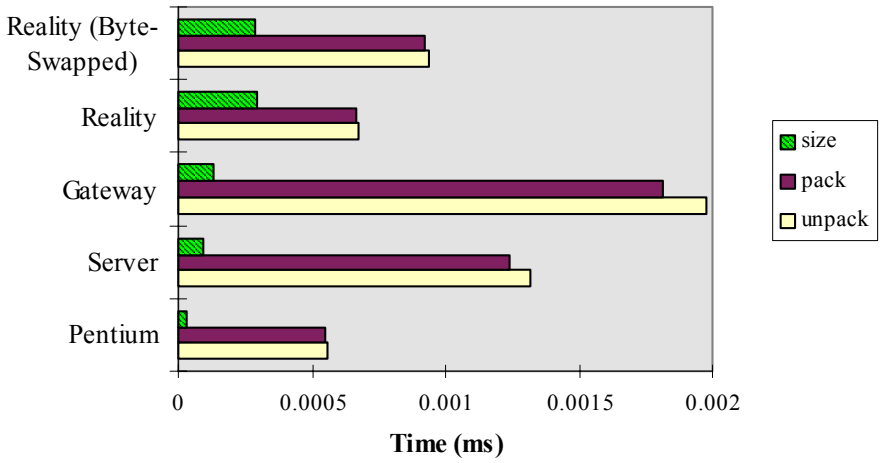


Figure 6.8 Costs of state sizing/packing/unpacking a Boolean/Integer/Real on all the test platforms.

When preparing state information for transmission its total size is estimated and a buffer is allocated into which the state is packed. The receiver of the state unpacks the transmitted buffer into its data structure. If the sender or receiver uses big-endian byte ordering, then a byte swapping operation is performed when packing or unpacking respectively. Figure 6.8 shows the performance of each of the test platforms when the three state encoding operations are performed on primitive types of the same size. Although Reality must always byte-swap its state, the performance of these operations without byte-swapping is shown for comparison purposes.

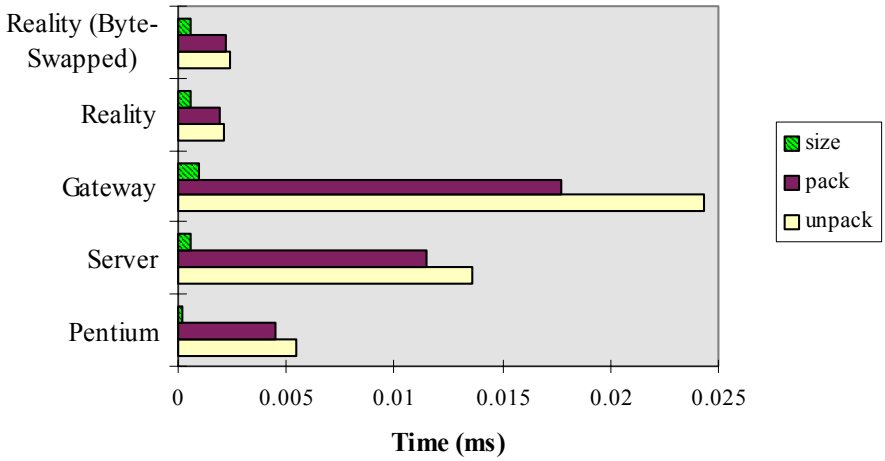


Figure 6.9 State encoding operation overheads for a String of 40 characters.

The time it takes to perform these same operations on a string with 40 characters is about 10 times slower than for the other primitive types (Figure 6.9). The extra time is consumed by the larger amount of data that must be copied into the buffer. There is no real difference between the performance of the byte-swapped operation and the normal version because character strings are not swapped in any way, only the integer that is used to hold the size of the variable length string.

6.3.6 Summary of UML Analysis

The amount of memory used by the interpreter is just as important as how fast it can interpret and execute UML code. The size of the textual UML definition is of interest since it may be sent between processes and thus affects communications performance. Whilst compression techniques can greatly reduce the space used by such descriptions, the computational overhead is prohibitive unless specialised hardware is available to accelerate the compression and subsequent decompression process. A compromise could be the transmission of tokenised code but this would reduce readability.

The cost of interpreting such definitions was presented in the previous sub-sections. Not only may the computational cost of managing the interpreter's internal data structure be estimated, but also the memory it occupies by applying the simple equations and empirical data in sections 6.3.2 and 6.3.3. It has been shown that there is a simple relationship between the time taken to process a component's state, its size and its structure. Such a relationship enables predictions to be made about the time required to manage state information.

6.4 PML

Performance evaluation of the PML can be conveniently broken into two parts: message transmission and message reception. Although the time taken to send a message is somewhat dependent on the processing done at the receiver, they can, for the most part, be treated separately. All of the charts in this section are based upon message size and therefore have

only been calculated up to the largest message size currently supported: 20 Kbytes[3]. In most cases, the performance of only one platform will be presented although the full suite of benchmarks were executed on all platforms. The equivalent graphs for the other platforms can be found in Appendix D.

This section examines the performance of both the QNX IPC and TCP/IP mechanisms as utilised by the PML. Only Gateway supported TCP/IP under QNX, this is unfortunate because it is also the slowest of all the test platforms. However, the relative performance of these two mechanisms can still be compared.

6.4.1 Transmission

Each communication mechanism shares a common need for a separate mailer process used purely for message transmission. In addition to the general cost of each IPC mechanism, the impact of communications to the mailer and the effect of transmission over Ethernet are examined.

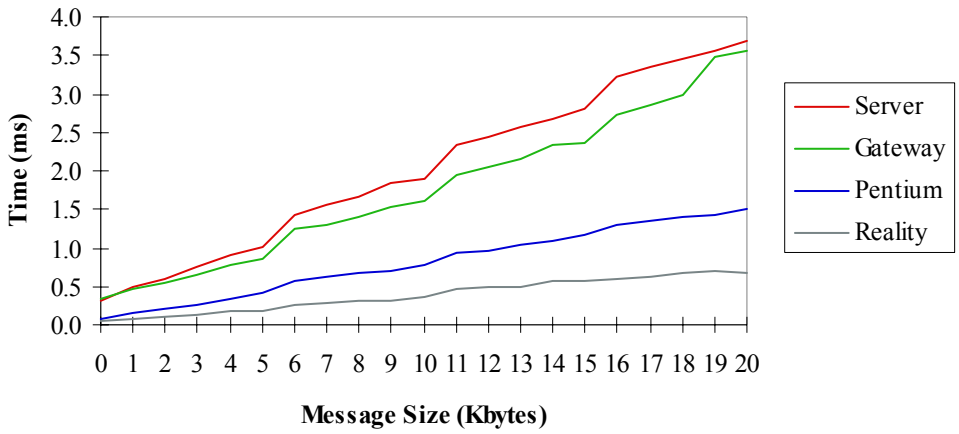


Figure 6.10 Relative overheads imposed by a pipe on each test platform.

6.4.1.1 Pipes

Figure 6.10 shows the time taken to transfer messages with different sizes along pipes on each platform used in the evaluation. Under QNX a pipe has a buffer size of 5 Kbytes, therefore when a message length exceeds a multiple of this buffer size, an extra `read()` system call is required. This extra operation is reflected in the chart as small jumps in transfer time at 5, 10 and 15 Kbytes. Despite having the faster CPU clock speed, Server has the worst performance. This can be attributed to having a slower internal bus speed than Gateway, whereas Pentium benefits from having a much faster CPU. IRIX uses a pipe buffer size of 10 Kbytes but the test results are too noisy to identify the relevant shifts in performance.

[3] This is an arbitrary limit imposed in the prototype and does not reflect an operating system limitations.

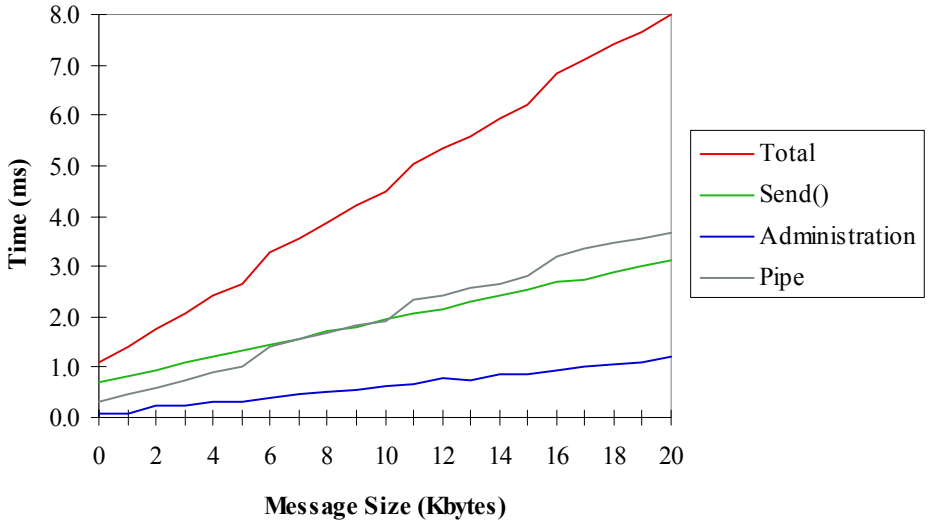


Figure 6.11 Deconstructed PML overheads for sending a message under QNX (Server).

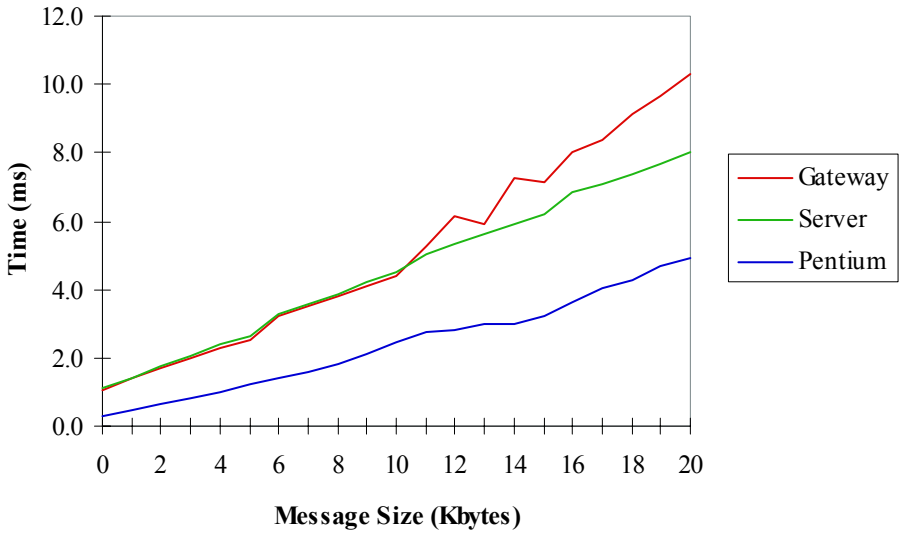


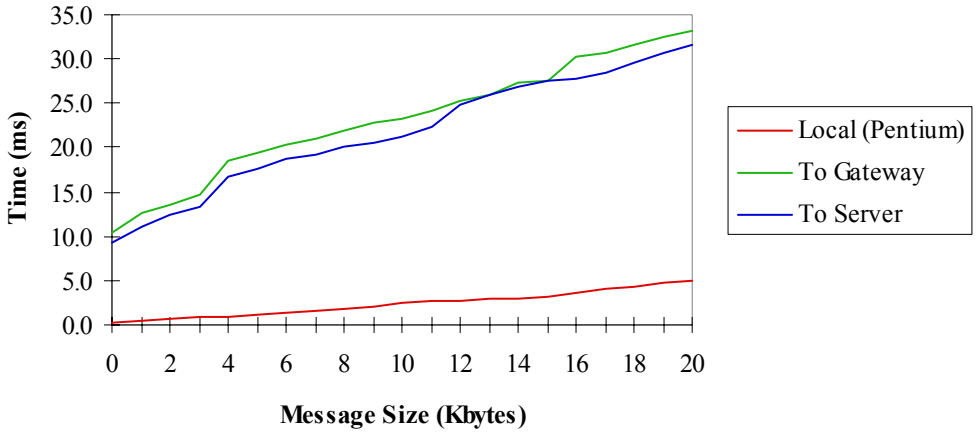
Figure 6.12 PML message transmission latency between local processes on the QNX platforms.

6.4.1.2 QNX IPC

Figure 6.11 shows a simple breakdown of the tasks performed by the PML in order to send a message using QNX IPC. The administrative overheads include filling the transmission buffer and, for remote communications, establishing and destroying a virtual circuit. The time taken to complete the actual `Send()` system call is also shown, including the time that the remote PML needs to receive the message and unblock the sender. All message sends must

be sent to the mailer via a pipe (section 5.3.5.4); the delay caused by this is also shown and is added to the other overheads to produce a total send time. The proportion of time used by each of these tasks is similar for each platform.

a)



b)

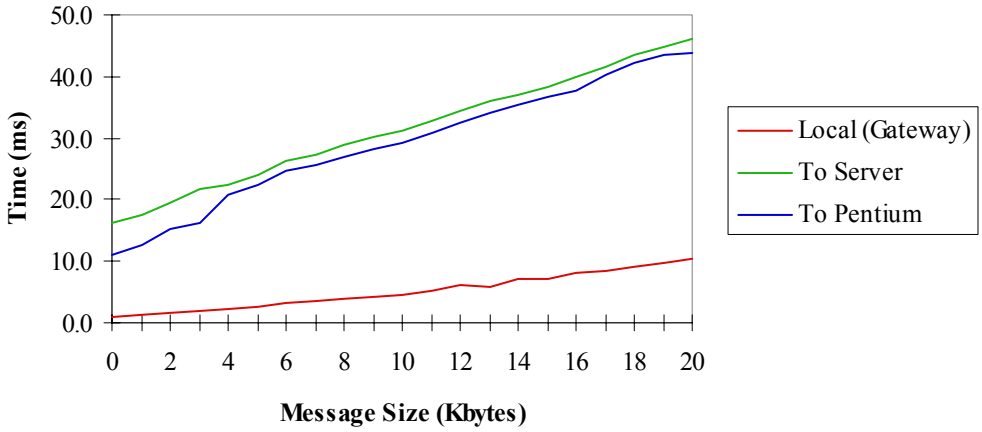


Figure 6.13 PML message transmission latency between remote processes on: a) Pentium and b) Gateway.

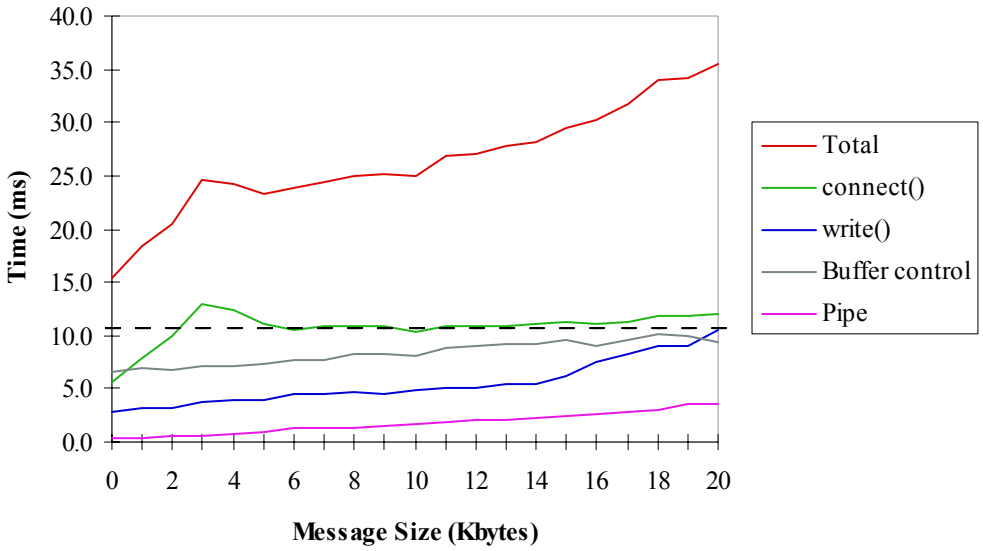
6.4.1.3 Latency

A comparison of the different QNX platforms used to run the prototype and their impact on message transmission latency is shown in Figure 6.12. The plotted data includes the latency introduced by the pipe. Figure 6.13 shows the difference in latency between local and remote inter-process communications. Unsurprisingly, on Pentium, communications with Gateway have the highest latency since it has the slowest processor. At the other extreme, when examining the same properties on Gateway, the longest delay is experienced when communicating with Server (Figure 6.13b). This result is forseeable since it is the slowest combination of CPUs within the three systems.

There is, therefore, a large difference between the latency experienced when sending a message to a local process and one on a remote node. On Pentium this magnitude ranges from

6-30 times longer for a remote communication, whilst Gateway experiences anything from 4-15 times greater delay.

a)



b)

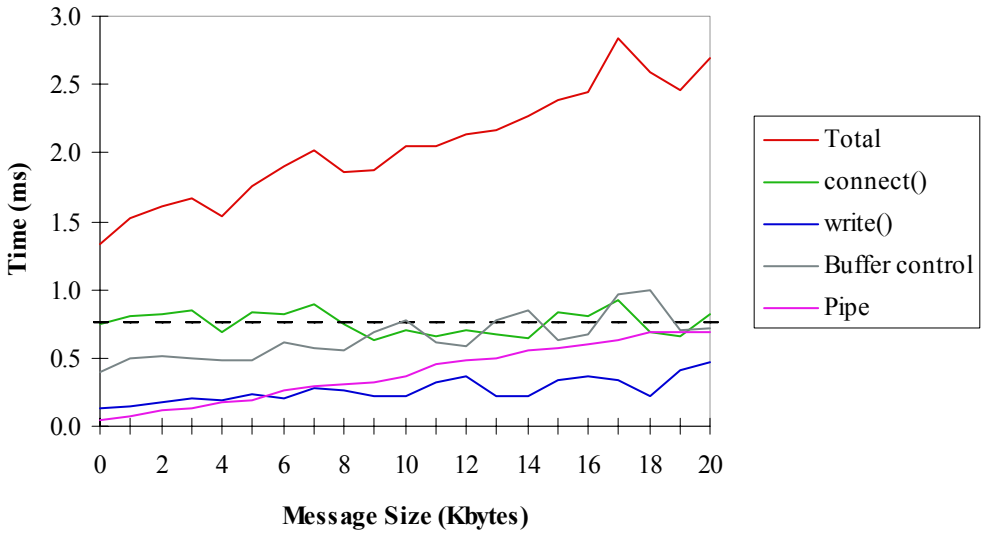


Figure 6.14 PML message transmission times for TCP/IP: a) under QNX (Gateway); b) IRIX (Reality).

6.4.1.4 TCP/IP

Figure 6.14 shows the time taken by each of the main stages to send a message using TCP/IP (under both QNX and IRIX) to another process on the same node and an absolute total which includes the pipe overhead. Establishment of a connection to the destination process is the most expensive stage: approximately 11 ms on Gateway and 1 ms on Reality (shown as dashed lines). The default TCP transmit buffer size under QNX is 7300 bytes and the default receive buffer size is 8192 bytes, therefore the TCP buffers were set to accommodate the

largest message size under QNX to avoid unnecessary message segmentation. This action alone accounts for around 4 ms of the total time required for buffer control. The IRIX buffer sizes default to 64 Kbytes and were not modified for the test.

The large performance difference between the QNX implementation of the TCP/IP protocol stack and its own proprietary IPC mechanism is shown in Figure 6.15. The reasons for poor TCP performance are discussed in section 6.4.4.

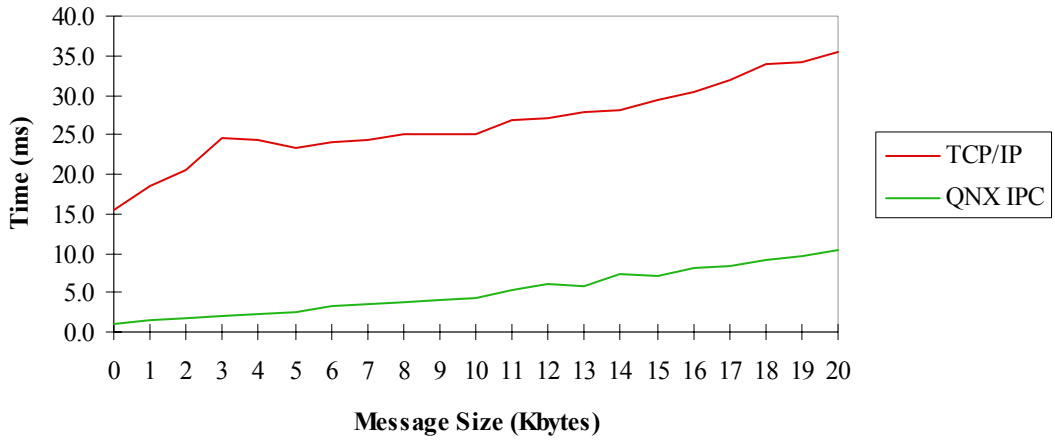


Figure 6.15 Effect of protocol on local PML transmission time (Gateway).

6.4.2 Reception

The tests used in this section are based upon the same methodology used in the previous section for message transmissions and are decomposed into their constituent tasks. Calculations are simplified, however, since the latency introduced by a pipe is not present when receiving a message.

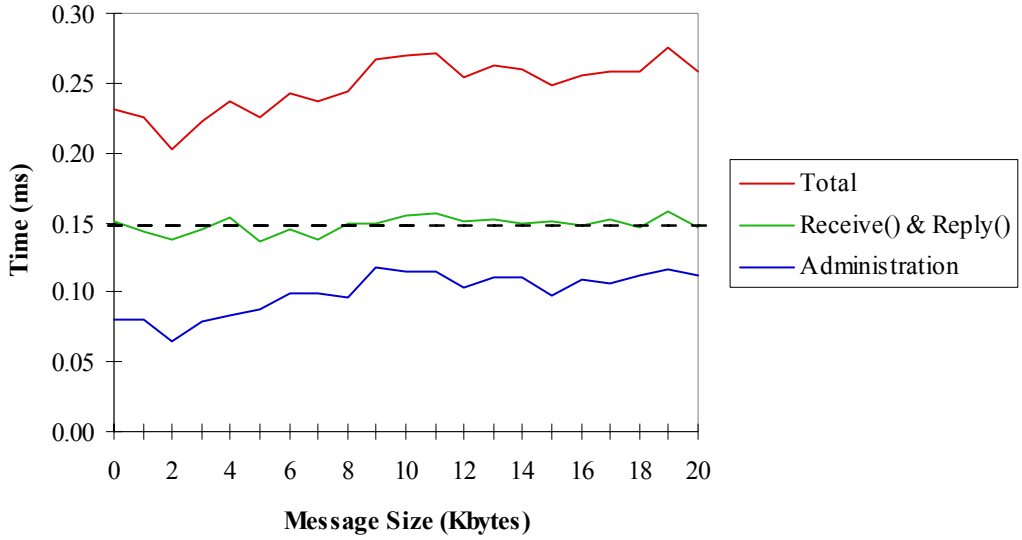


Figure 6.16 Breakdown of a PML message receive under QNX (Gateway)

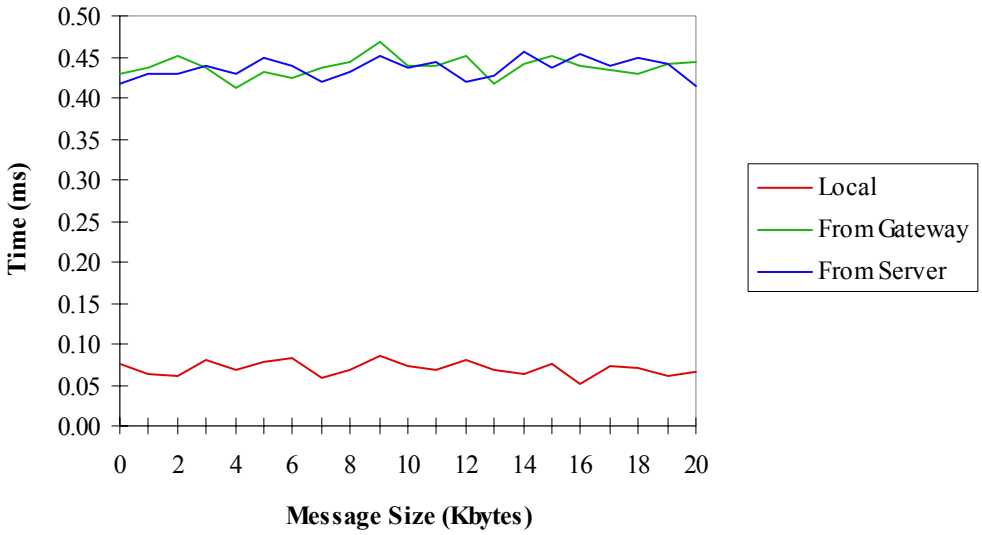


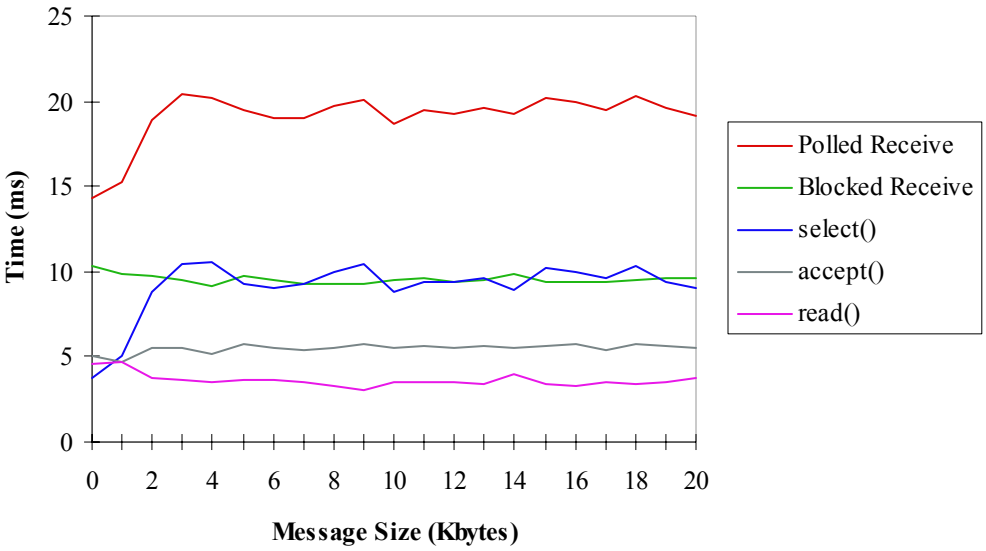
Figure 6.17 Comparison between receiving messages from local and remote processes (Pentium).

6.4.2.1 QNX

There are three basic tasks that are performed when receiving a message using QNX IPC: actual message reception into the receive buffer, unblocking the sender and extracting the message from the buffer. To minimise the transmission latency, the sender is unblocked directly after the buffer has been filled. The average time for this sequence of events is shown by a dashed line. Figure 6.16 shows how much of the total receive time is used by the administration overheads. A slight trend towards longer durations is visible in the administration tasks as the message size increases.

In a similar manner to message transmission, receiving a message from a remote process takes a lot longer than from a local process (Figure 6.17): on Pentium approximately 6 times longer.

a)



b)

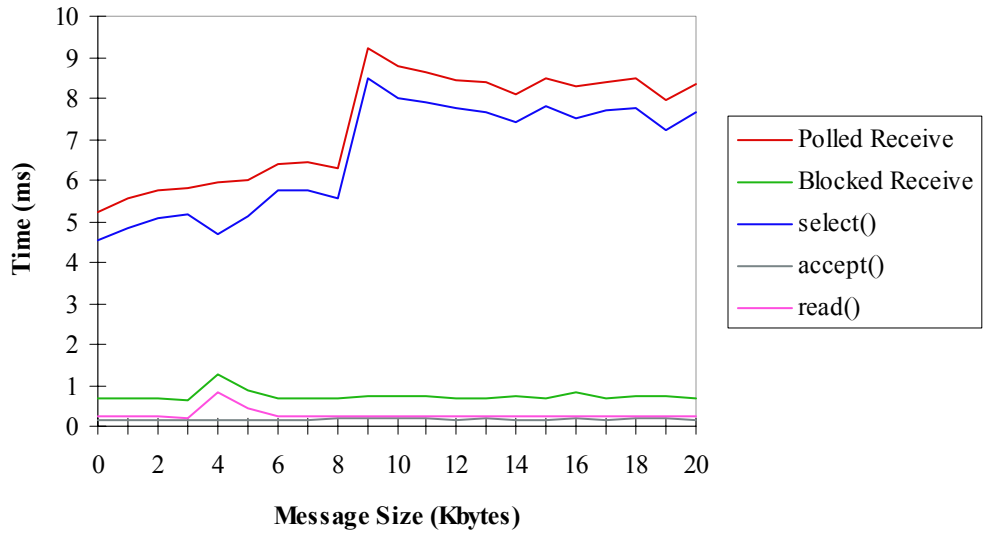


Figure 6.18 PML message reception using TCP/IP: a) QNX (Gateway); b) IRIX (Reality).

6.4.2.2 TCP/IP

Figure 6.18 depicts the duration of the major stages required to receive a message using TCP/IP through polling and blocking. When blocking for a message, the `accept()` call is issued immediately; when a connection is made, the message is spooled into the receive buffer and then the message is extracted from it. For a polled receive, the `select()` system call is used to check for pending connections and `accept()` is only called when there is a connection waiting. The chart deceptively shows that a blocked receive is faster than a polled

receive since it does not include the (potentially very long) period when the process is waiting. Under QNX, the receive buffer was increased in size in order to accommodate the largest possible message. Again we see that Reality outperforms Gateway by approximately 10 times for a blocked receive and 2-3 times for a polled receive. The unusual sharp decrease in performance experienced at about 9 Kbytes with IRIX TCP/IP is consistently repeatable. The only explanation that the author can offer is that this is the result of some internal buffering in the IRIX socket daemon and may be connected to the poor TCP/IP performance experienced (section 6.4.4).

6.4.3 Throughput

A useful metric is the amount of data that can be transmitted in any given period of time - throughput. This section discusses two forms of this metric: *local throughput* which refers to data transfer within a machine and *network throughput* which refers exclusively to data transfer between machines.

6.4.3.1 QNX

Comparison of local throughput is straight forward when all platforms use the same operating system. The maximum throughput at a given point can be calculated as follows:

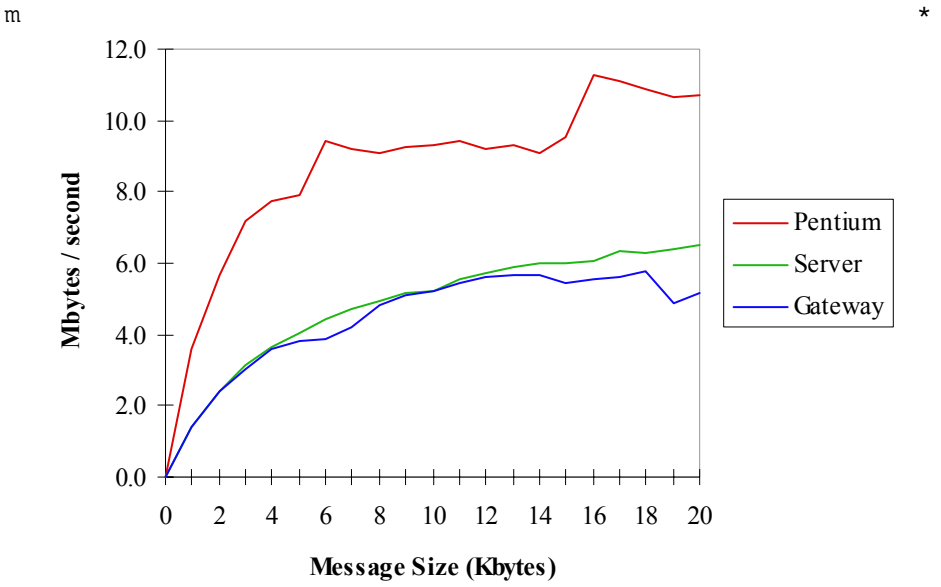


Figure 6.19 Maximum local throughput within each node using QNX IPC.

Using QNX IPC, it is necessary to subtract the time it takes the receiver to unblock the sender from the actual send time (not including administration overheads). Then it is just a matter of converting the result into Mbytes per second. Figure 6.19 shows the maximum amount of data that can be sent within each system based upon message size. If throughput was limited by bus speed we would expect to see Gateway slightly outperforming Server and an increased throughput for Pentium with its PCI bus. As it stands, however, internal throughput seems to be compute bound. The actual throughput will be less if the sender and/or receiver are not getting as much CPU as they need.

All of the QNX platforms share the same physical LAN and each has the same make of network card connected to the same type of internal bus. Therefore, it should be possible to estimate maximum network throughput using the same technique used for local throughput. Figure 6.20 represents the estimated maximum network throughput between each possible node pair. The calculation was performed each way on the link, e.g. Pentium to Server and Server to Pentium, and the result averaged to simplify this chart. The results reinforce the conclusion that throughput is compute bound: the two fastest machines have the highest throughput, followed by the fastest and slowest and then the two slowest machines.

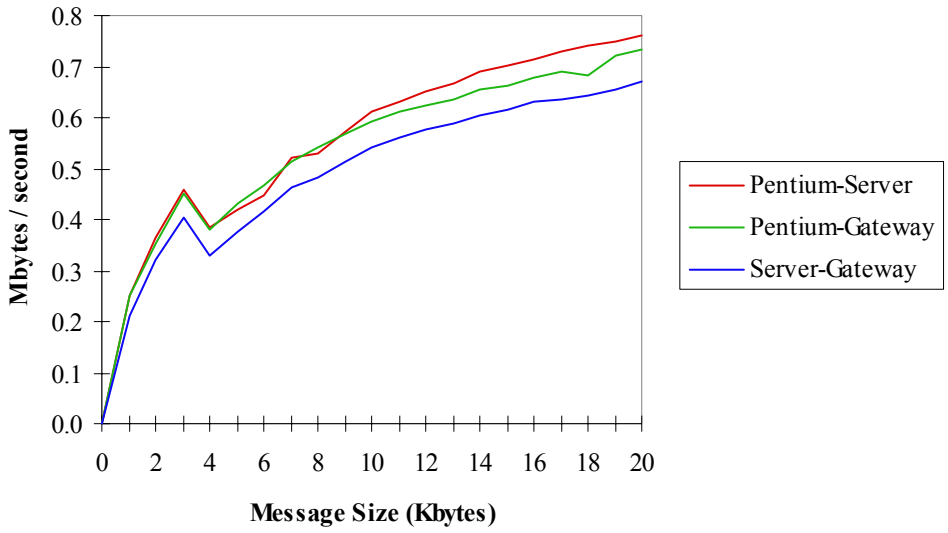


Figure 6.20 Maximum QNX network throughput between node pairs.

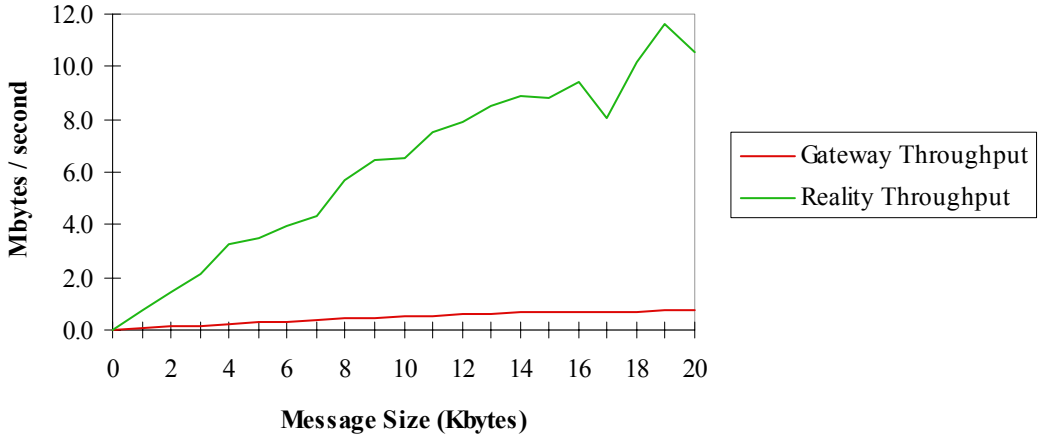


Figure 6.21 Maximum TCP/IP message passing throughput for each platform.

The uncharacteristic drop in performance when the message size reaches 4 Kbytes is caused by a large number of out-of-window collisions being generated by faulty Ethernet cards. In fact, throughput should rise dramatically as message size increases and start to level out at around 6 Kbytes. This problem had not been noticed until these tests were run.

The largest message transfer gives Pentium a network throughput of 0.762 Mbytes/sec which may also be expressed as 6.096 Mbps. On a 10 Mbps Ethernet network this is a high utilisation rate which may be attributed to QNX's lightweight protocol and few collisions due to the controlled manner in which the tests where executed. This figure is, in fact, 0.1 Mbytes/sec lower than the manufacturer's own performance data for a 20 Kbyte message and is almost certainly due to the aforementioned problem.

6.4.3.2 TCP/IP

The local message passing throughput for QNX TCP/IP is on the same scale as that of QNX IPC network throughput. Reality, however, matches the top QNX message passing performance (Figure 6.21). A meaningful value for the network throughput between these two machines could not be obtained because they are located two miles apart and are separated by two Ethernet LANs, a large FDDI MAN and many routers. The traffic on these networks is generated by machines scattered throughout the university.

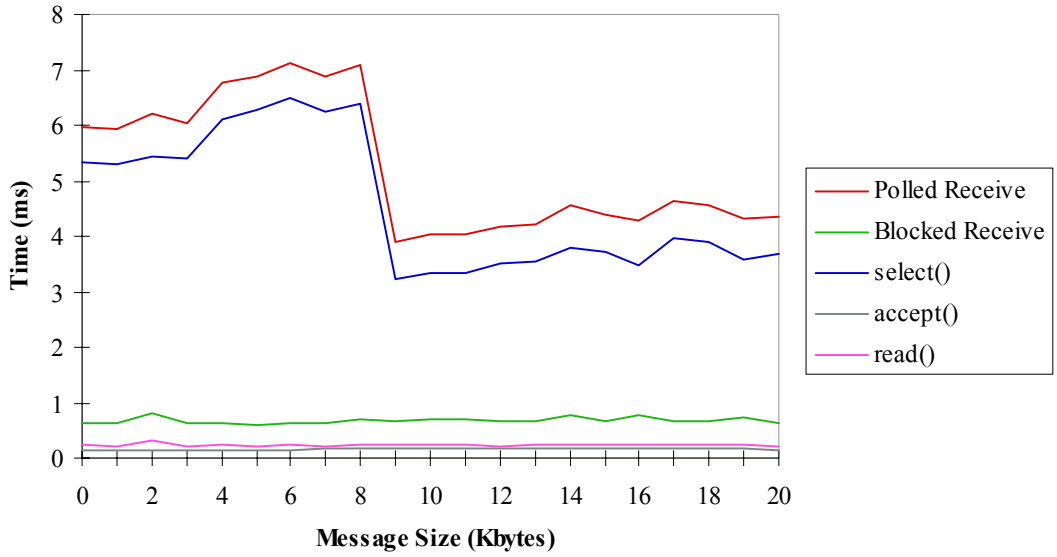


Figure 6.22 PML message reception test for TCP/IP on Reality with TIME-WAIT build-up.

6.4.4 TCP/IP Performance

The way that TCP is used by PML has highlighted a problem with this protocol. A connection passes through various states during its lifetime, the last of which is TIME-WAIT. The connection spends long enough in this state to ensure that the remote end has received the acknowledgement of the connection termination request and that all segment[4] duplicates have expired (Postel, 1981a). This period is twice the Maximum Segment Lifetime (MSL) which is the time a TCP segment can exist in the internetwork system. MSL has been arbitrarily

[4] The user message data is broken into segments by the TCP when sent along a connection.

defined as 2 minutes although, as noted by Jacobson *et al.* (1992), TIME-WAIT has more to do with the round-trip time for the connection than anything else. Regardless, if TIME-WAIT is not long enough it is possible for old duplicates to infect a new connection (Braden, 1992).

Jacobson *et al.* have noted that this state could cause an indirect performance problem if an application repeatedly closes one connection and opens another at a very high frequency. The current limit of available TCP ports on any host is 2^{16}. PML establishes a connection *every* time a message is sent, consequently there is a rapid build-up of connections in the TIME-WAIT state. For simulations with many entities this can soon produce thousands of connections in the time-out phase. The results shown in Figure 6.18 were obtained by ensuring that the benchmark for each message size started when there were no connections still timing out. Figure 6.22 shows what happens if connections are made with others still in time-out. When the message size is small the benchmark program has a short execution but creates a lot of connections. Up until around 8 Kbytes this happens at a rate faster than time-outs occur, but afterwards more connections time-out than are established which results in increased performance. Under QNX the time-out period is around 30 seconds whilst IRIX uses a period around twice that which means that this problem is less pronounced with QNX.

The only way of improving performance using TCP is to maintain fixed connections between key processes but at the price of increased memory and computational overheads on the part of the PML (section 5.3.5.3). Before any decision is taken on whether or not to pursue this solution, it would be prudent to investigate the potentially more rewarding problem of a reliable datagram service (section 6.6.3).

6.4.5 PML Summary

The tasks of sending and receiving messages using two IPC mechanisms have been broken down into their constituent parts and analysed. QNX IPC is very lightweight and subsequently outperforms TCP/IP when running under QNX. The faster processing power available to the IRIX implementation shows that this protocol can be used in systems of this nature. However, its performance is rather unpredictable, especially when there is a high connection turnover rate. It would seem, therefore, that TCP/IP is best used for communications between nodes in a USS and that an alternative local IPC mechanism is used, e.g. based upon shared memory.

When lightweight threads become readily available it will remove the need for the physically separate mailer process and thus the latency introduced by the pipe. This would improve transmission times at the most by between 0.5 ms and 3.5 ms depending on the platform.

The dramatic difference between message passing performance locally and remotely using QNX IPC was shown in Figure 6.13. This is due to a throughput difference of over 10 times and emphasises the importance of reducing to a minimum the amount of data that is sent between machines. In this test case the machines were only located 1 metre from each other, if they had been further apart, e.g. separated by routers, the results would have been even worse.

6.5 Simulation Execution

A UM provides a number of services, most of which serve to progress the simulation as fast as possible. There are a number of factors that dictate performance:

- Number of entities.
- Number of managers.
- Number of monitored components.
- Frequency of state updates from each entity.
- Size of the state updates.

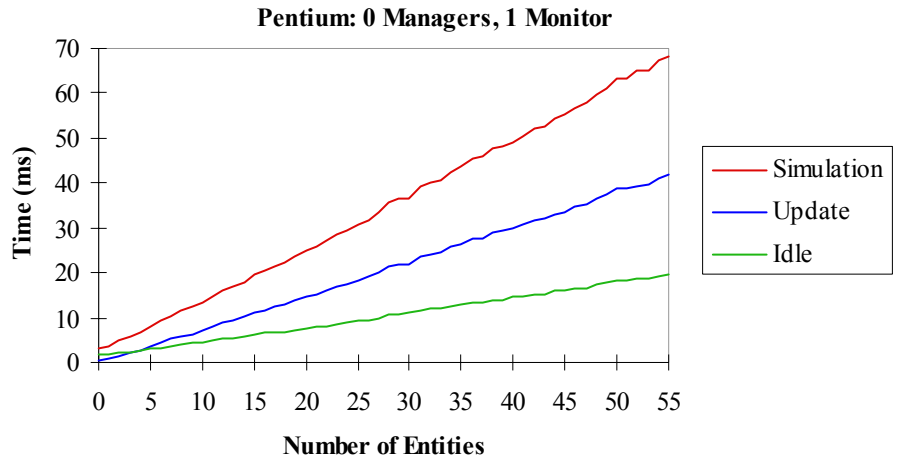
The contributions made by each of these factors is application dependent and can vary quite substantially. For example, an architectural walk-through may have a large number of entities but they will be predominantly static and therefore produce few state updates. On the other hand, a highly dynamic simulation such as birds flocking will require constant state recalculation. To fully explore all of the possible options would take a very long time and it is unclear what benefits such a varied and non-specific analysis would produce, therefore a more pessimistic approach has been taken.

The core sequence of events for one simulation cycle are as follows:

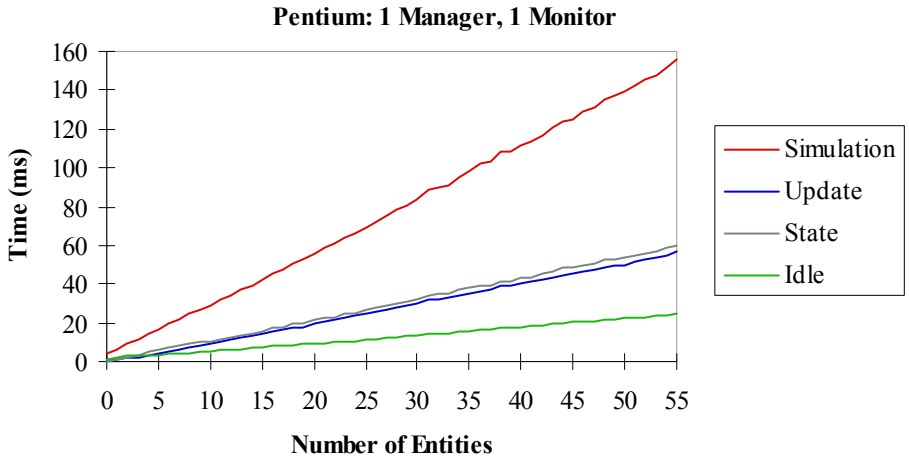
1. Send an *update notify* message to each entity and manager in the system.
2. Each entity sends its state updates to its local UM.
3. The UM forwards the state messages to interested managers (and other UMs).
4. When all state updates have been sent, each entity sends an *update complete* message to its local UM.
5. When all entities have completed the manager is informed and performs its processing.
6. The UM waits for all managers (and slave UMs) to finish their work before starting again at stage 1.

The factor that will have the most impact on performance is the amount of state updates that the UM must handle. This is directly related to the number of interested components and managers in the system. Through examination of a worst-case scenario, a more insightful and stable picture is presented of an architecture that attempts to reduce state flow as much as possible.

a)



b)



c)

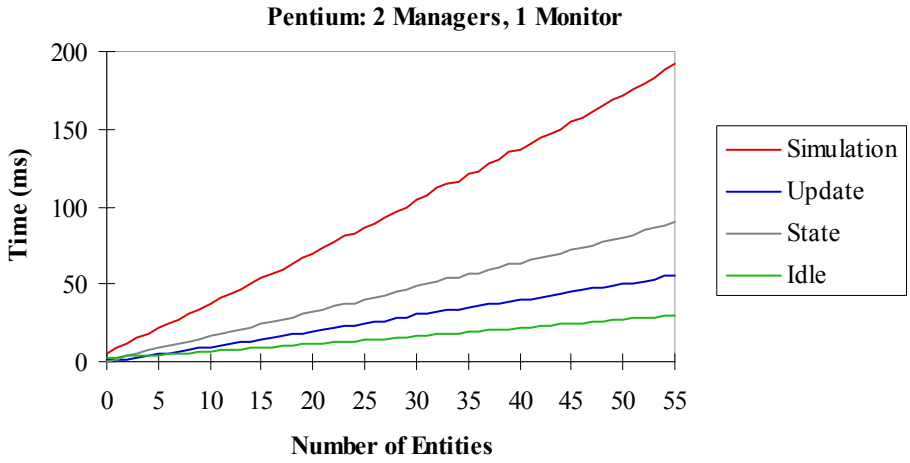


Figure 6.23 Activity breakdown of a UM when there is one monitored component and: a) no managers; b) 1 manager; c) 2 managers (Pentium).

To test the affects of state updates, each node was stressed using increasing numbers of entities, each of which modified its state every simulation step and was monitored by an increasing number of managers. The majority of the charts in this section show the duration

of the simulation step as it is affected by entity numbers. Under QNX, the number of entities that could be used was limited by the amount of memory available on each node. The memory consumption varied depending on the amount of state information each entity had and the number of managers in the simulation that were monitoring that state. The universe definition consisted of one and two properties of integer type for the one and two monitor cases respectively. The sizes of the actual messages used in the tests were very small, averaging < 100 bytes. This is because the state transmissions for the monitored components only contained data for one or two integers. If the simulation protocol overheads can be established then the impact that an increased state size would have on performance can be extrapolated from the knowledge of its structure (section 6.3) and the increased message sizes (section 6.4, section 5.3.3). The source code for the benchmark manager and entity used in the tests can be found in Appendix B.

Several configurations of a USS were examined:

1. Single node (all nodes were tested in this configuration).
2. Two nodes with the Pentium occupying the master node role and Server acting as the slave.
3. Three nodes - the same as configuration 2 but adding Gateway as another slave.

The test results obtained with these configurations are presented below and are followed by an examination of the entity migration mechanism.

6.5.1 Single Node

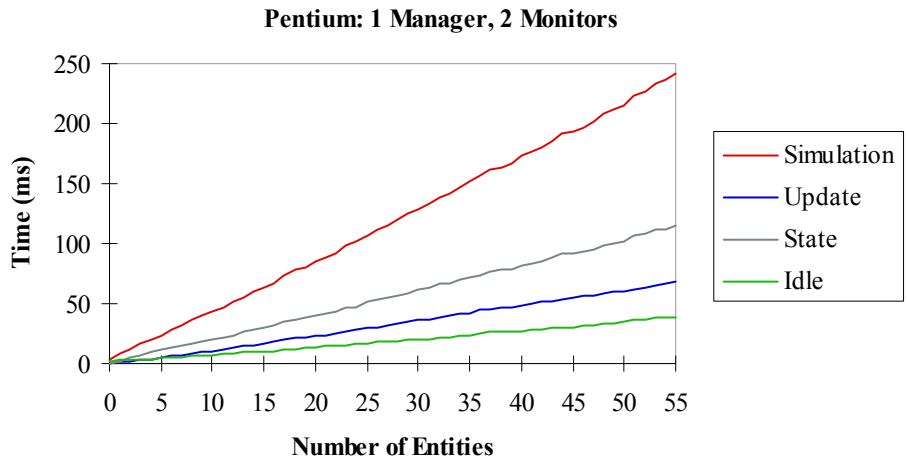
When there are no managers in a system, there is not a need for entities to send state updates. Consequently only update notification/complete messages are sent to the RM and update messages transmitted to each entity. The idle time shown in Figure 6.23 represents the time spent waiting for the entities to inform the UM that they have completed their update. When a special manager is introduced and registers interest in one component (1 monitor) a considerable time is spent relaying each entity's state to it. This does not, of course, affect the amount of time spent idle but does reduce it relative to the total simulation step duration. When there are two managers interested in the same component the state must also be sent to that manager (Figure 6.23c) which, in this case, means more time is spent relaying state than sending the update messages. The time needed to send the update complete messages increases slightly with each manager but is so small that it barely registers on these charts and is therefore not shown.

Figure 6.24 shows equivalent charts where there are two monitored components. When the case with a single manager and two monitors is compared with that of a single manager and one monitor, it is clear that the time spent sending state information has doubled. The same is true for the equivalent cases with two managers.

Two different perspectives on these results are shown in Figure 6.25: firstly in terms of simulation steps per second and, secondly, as a workload relative to the case with no managers. An extra case is presented here, that of three managers and 1 monitor whose

performance is matched by the 1 manager, 2 monitors case. This may be explained by examining the messages sent in each circumstance.

a)



b)

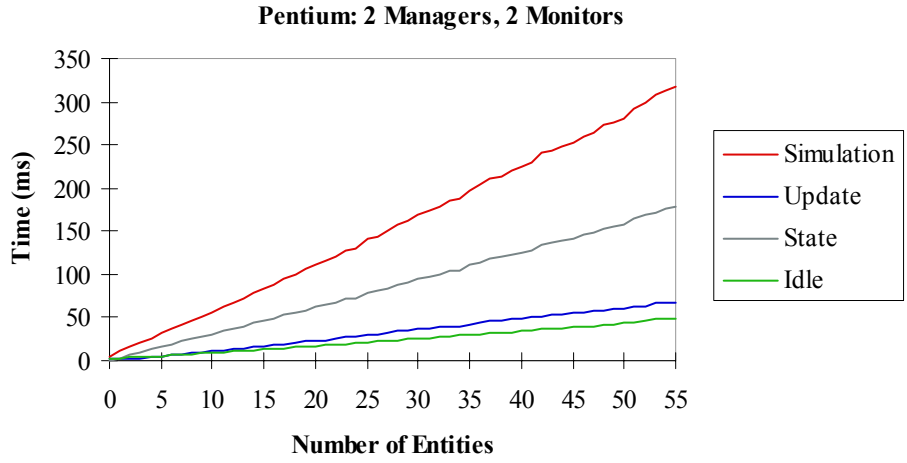


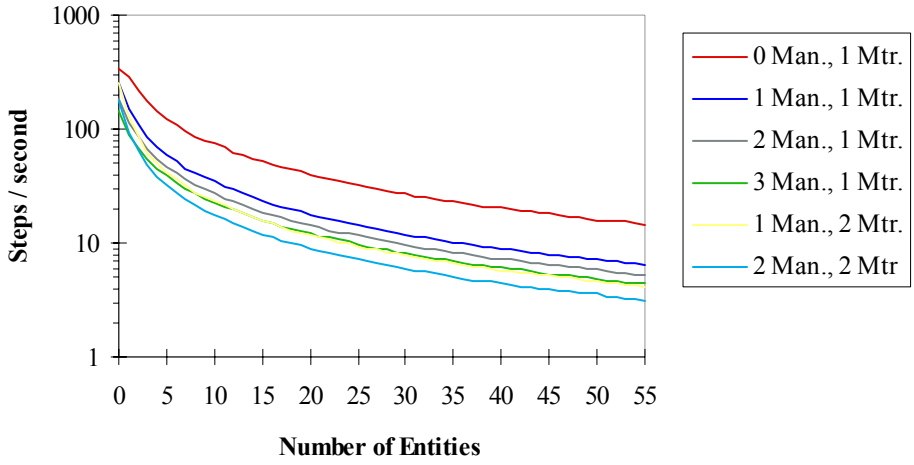
Figure 6.24 Activity breakdown of a UM when there are two monitored components and: a) 1 manager; b) 2 managers (Pentium).

If there are *x* managers, *y* monitored components and *z* entities, then *y*z* state updates are sent by entities to the UM and *x*y*z* state messages received by managers in total each step. For 10 entities this results in 10*1 messages originating from entities and 3*10*1 messages sent to managers in the former case - a total of 40 message transmissions. Applying the equations to the latter case, 20 state messages are sent by entities and 20 messages received by the manager. Therefore the same amount of bandwidth (40 state messages) is being used every simulation step resulting in the same simulation rate. The slight performance discrepancy visible in the charts can be attributed to the different numbers of update notification/complete messages that are sent in each case and the difference in total manager overheads. In this case, state transmissions are the largest performance limiting factor.

The UM's idle time is the sum of the total time spent polling for an incoming message (because there are still pending internal events in the action queue) and the time spent blocked, waiting for a message since there is no other work to do. Figure 6.26 shows this idle period as a percentage of a simulation step (which gets longer as the number of entities

increases). For small numbers of entities the amount of time spent idle is high but it soon settles into a consistent rate as the number of entities and system workload increases. When there is no state to forward, the UM is idle for around 30% of the time, but when there is one or more managers in the system, the UM idles approximately 16% of the time.

a)



b)

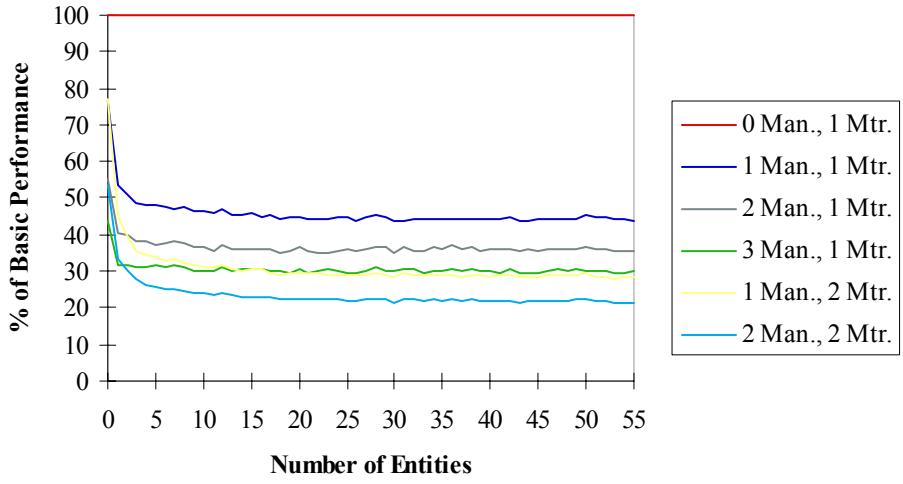


Figure 6.25 Effects of various factors on simulation rate: a) steps per second; b) percentage of basic performance (Pentium).

All of the results presented here are from the tests performed on the Pentium platform. The results for the other platforms have the same relative proportions but indicate a lower performance. Figure 6.27 shows how the baseline UM performance (0 managers, 1 monitor) compares with the equivalent configuration on Gateway. Pentium consistently performs on average 3 times faster than Gateway. Server's results (not shown) are very similar to that of Gateway's, reflecting their comparative computational power. Complete charts may be found in Appendix E.

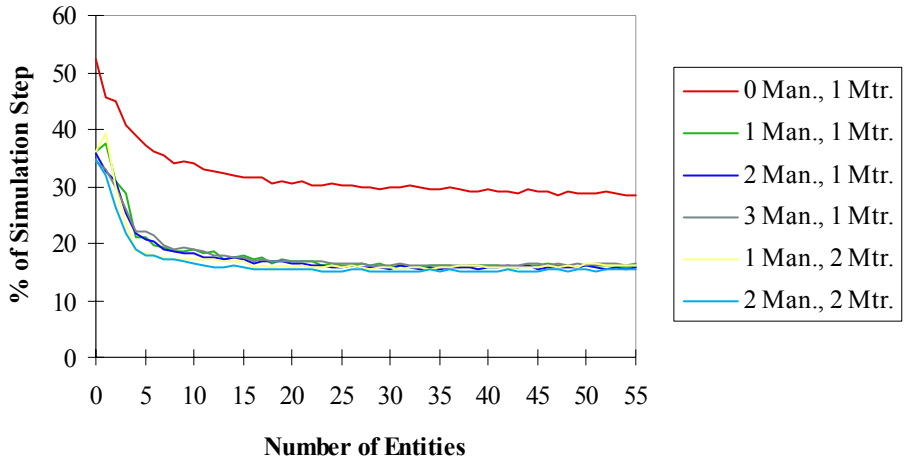


Figure 6.26 Percentage idle time in the UM for each test case (Pentium).

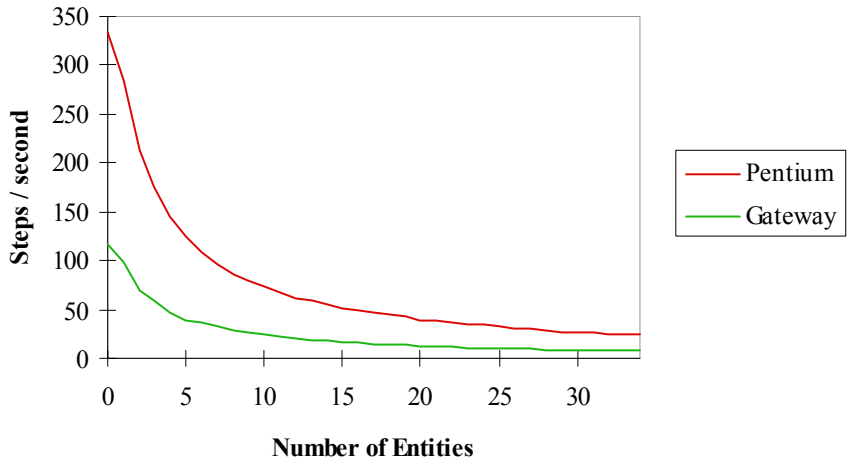
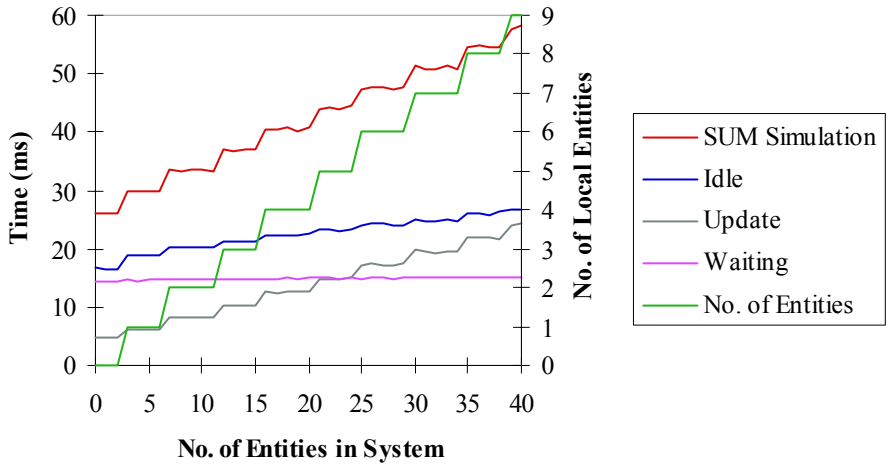


Figure 6.27 Comparison of Pentium and Gateway baseline performance.

6.5.2 Two Nodes

To examine the performance effects of a multi-node USS, the same tests used in the single node trials were repeated with the entities distributed amongst the nodes. The decision of whether to allocate an entity to one node or the other was based upon a CPU rating derived from the single node results obtained previously. For example, in the case with no managers, the total simulation time for 31 entities on Pentium is ~38 ms, whereas this same time is used by 9 entities on Server. This would give a CPU rating for Pentium of just over 3 times that of Server's, a figure backed up by the CPU performance figures given in Table 6.1.

a)



b)

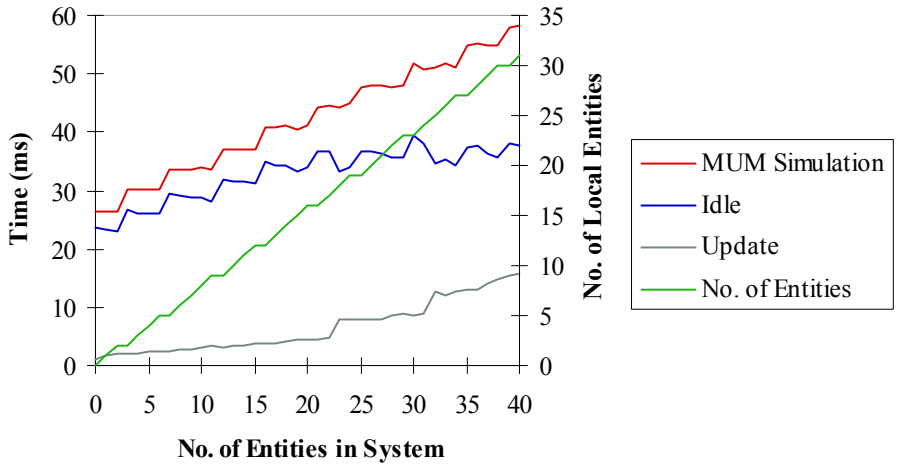


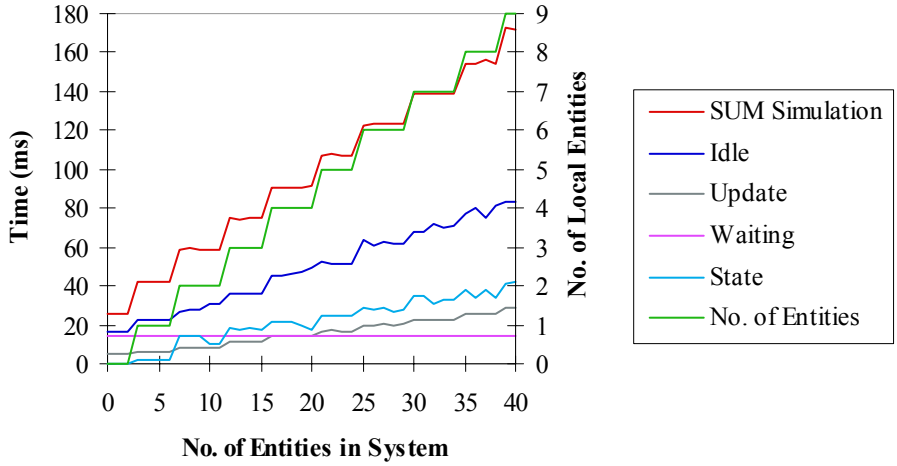
Figure 6.28 Activity breakdown of UMs in a master-slave configuration with no managers: a) SUM (Server); b) MUM (Pentium).

In this and subsequent multi-node tests the fastest node was used to run the MUM, the activity breakdown of which may be found in Figure 6.28b. The total simulation time for each node is identical since the SUM must wait for the MUM to send it an update notification message before it begins each simulation step. The stepping effect is caused by the changes in entity distribution which is measured by the scale on the right hand side - at most 40 entities were used system-wide. The somewhat irregular shape and downward tilt of the steps is a reflection of the error in the distribution algorithm. That is, whereas an optimum distribution may require fractional parts of an entity to be distributed in order to keep the workload exactly balanced, only whole entities can be moved.

A large portion of the SUM's time is actually spent waiting for the MUM to start the next simulation step - 14 ms in this case. The SUM notifies the MUM that it has completed its processing for that step and, when all the MUM's local processes have finished as well, the MUM sends the next update complete message. The waiting time is therefore the sum of two message transmission latencies and some processing with the exception of one condition. It is true that a significant portion of the idle time can be attributed to the waiting period. However, it is possible for the SUM to wait for a period greater than its idle time if the SUM

should be starved of CPU - a situation that may occur in a heavily loaded simulation. This aside, if the waiting period is subtracted from the simulation time series, the product is the equivalent of a single node simulation.

a)



b)

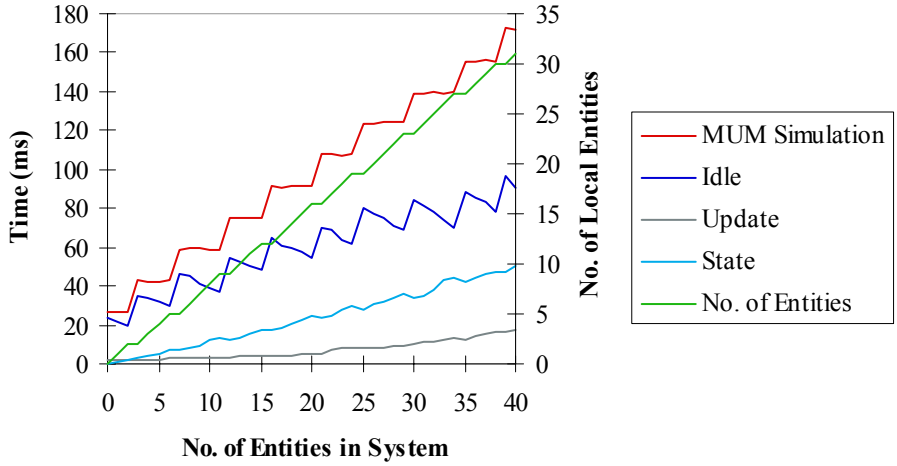
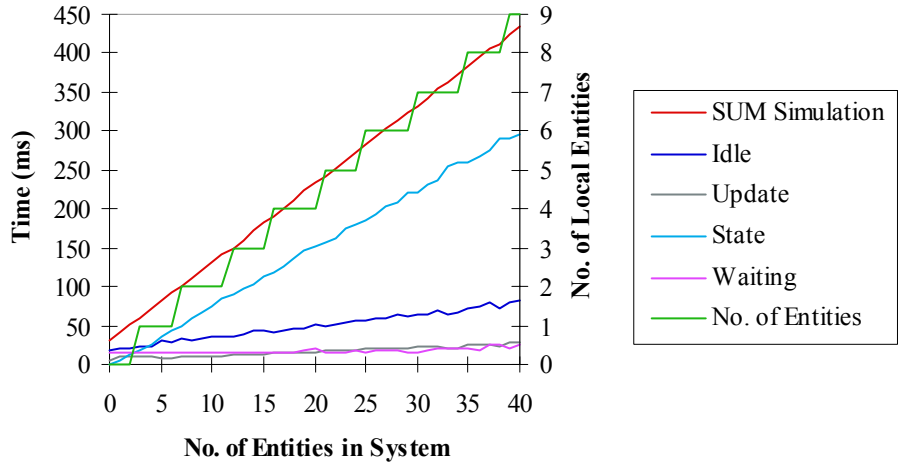


Figure 6.29 Activity breakdown of UMs in a master-slave configuration with 1 manager on the master node: a) SUM (Server); b) MUM (Pentium).

The same test conditions were used to introduce a manager on the node with the MUM and the largest number of entities (Figure 6.29). This reduces the state updates sent over the network to a minimum, i.e. the few entities on the SUM send them to the MUM. Initially, when there are no entities on the slave node, performance is identical to the previous case. However, as soon as an entity is allocated to the slave the latency of a state update is incurred. This, added to the additional message processing on both nodes significantly increases the duration of the simulation step.

a)



b)

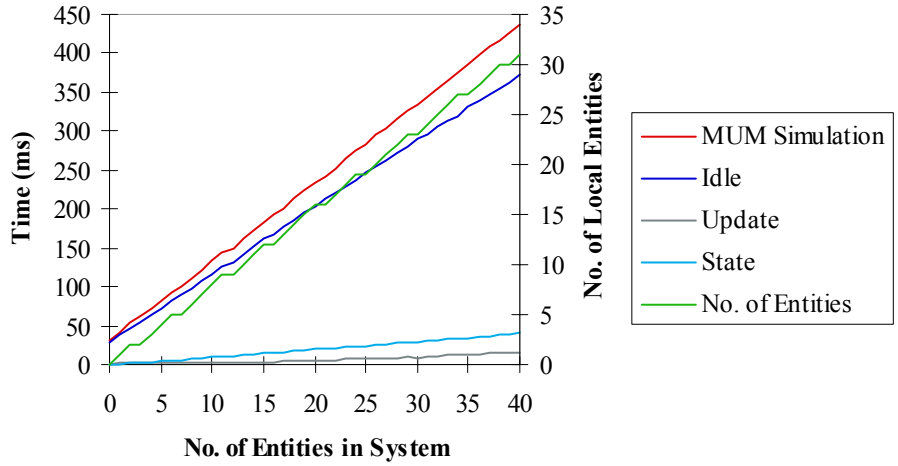


Figure 6.30 Activity breakdown of UMs in a master-slave configuration with 1 manager on the slave node: a) SUM (Server); b) MUM (Pentium).

The effect of placing the manager on the slave node rather than the master node may be seen in Figure 6.30. When there are 40 entities in the system, 31 state updates must be sent across the network to the SUM and then forwarded to the manager. The master's chart (Figure 6.30b) shows that the MUM spends most of its time idle, waiting for the slave to process all the state information. The chart is somewhat deceptive, however, since the state time does not include the message transmission latency which would put it close to the total simulation time and reduce the idle time appropriately. Introduction of a second manager on the master node increases total simulation time by about 100 ms (when using 40 entities) since state information is now also sent from slave to master.

It is clear, therefore, that not only is computational power an important consideration when distributing entities[5], but also the inter-node communication overheads and the location of

[5] Without resource dependencies, such as input devices.

special managers. With the technology used in this prototype, the network is by far the most limiting factor.

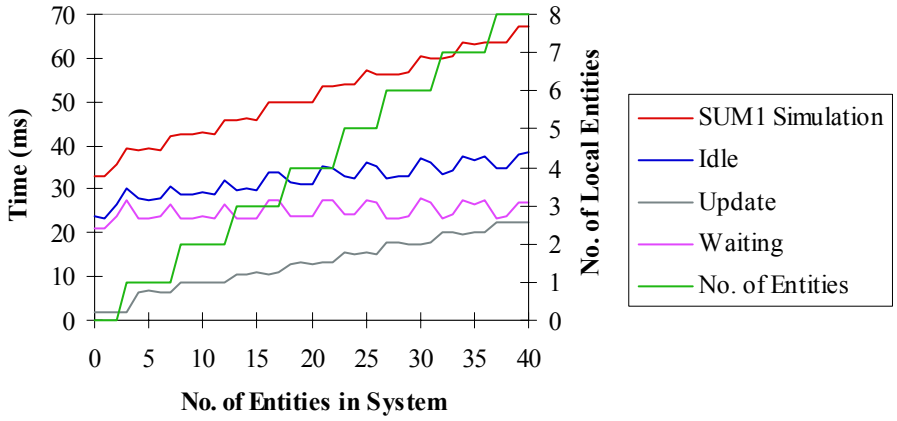
6.5.3 Three Nodes

Figure 6.31 shows the task durations on three nodes as the simulation is distributed amongst them. In contrast to the equivalent case with two nodes (Figure 6.28), the wait time has risen to 20 ms and the overall simulation time by 10 ms. The MUM's largest workload has been lightened by 8 entities which have been spread between the two SUMs. The added processing time incurred by the extra node has caused the MUM's increase in simulation and wait times.

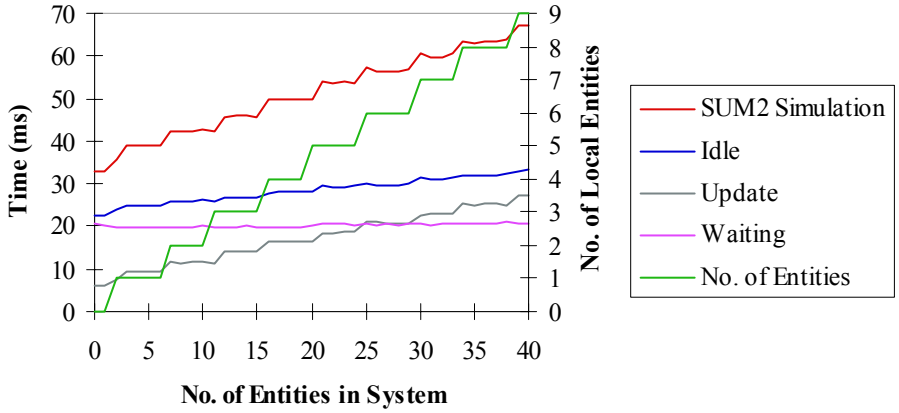
When a manager is added on the master node, the total simulation step time degrades to a maximum of 218 ms which is ~40 ms more than the equivalent case with 2 nodes (Figure 6.29). However, as Figure 6.32 shows, when the manager is allocated to a slave node, the overall system performance is identical to the master-slave case depicted in Figure 6.30. On an individual basis, the first SUM (on Server) is managing one less entity than previously which results in slightly lower state management times.

Unlike the other slave node, the SUM's waiting period is greater than its idle time, indicating that there are other processes on that node that have more urgent need of the CPU. The idle time is smaller because some of the time that the SUM would have spent idling was consumed when it was waiting for its next timeslice. Therefore, despite communication latency hindering performance, this three node configuration, with a manager on Server, is as efficient as the master-slave case presented in the previous section.

a)



b)



c)

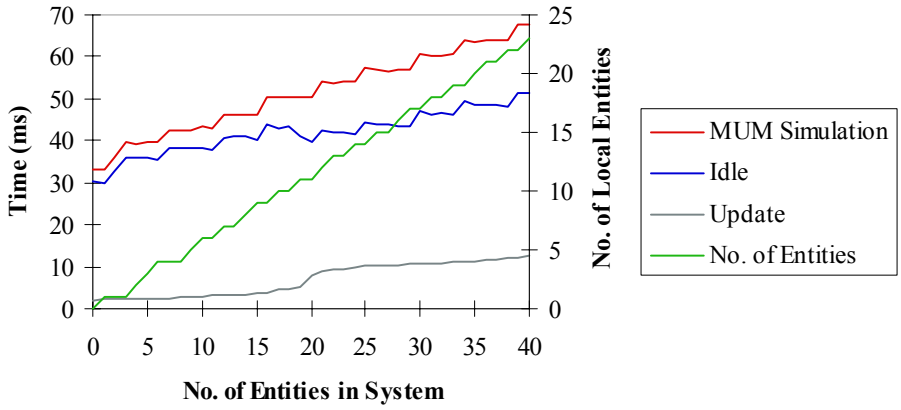
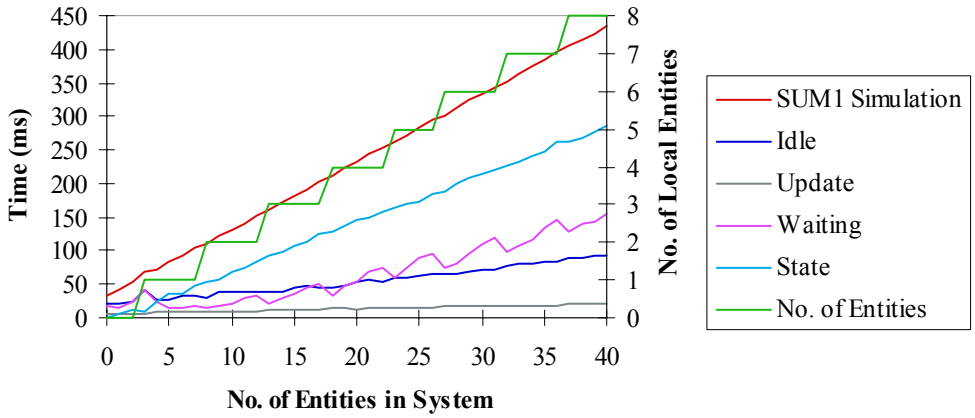
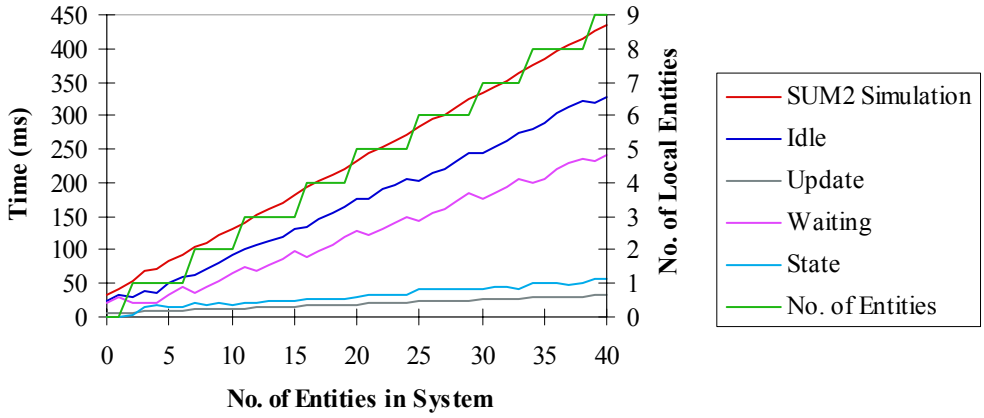


Figure 6.31 Activity breakdown of UMs in a master and 2 slaves configuration with no managers: a) SUM1 (Server); b) SUM2 (Gateway); c) MUM (Pentium).

a)



b)



c)

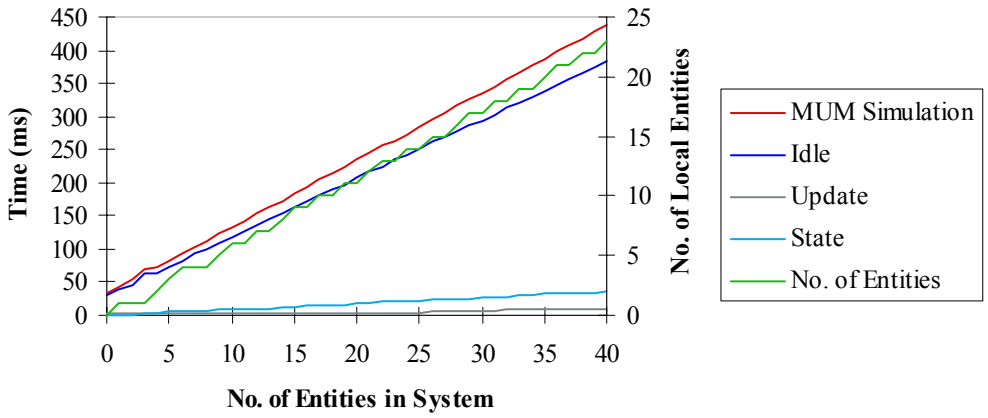


Figure 6.32 Activity breakdown of UMs in a master and 2 slaves configuration with a manager on slave node Server: a) SUM1 (Server); b) SUM2 (Gateway); c) MUM (Pentium).

6.5.4 Entity Migration

In order to demonstrate entity migration it is necessary to have some way of estimating resource usage for each entity. A measure is not meaningful unless it is measured with reference to a fixed time span, i.e. a simulation step. Since full scheduling and RM functionality had not been implemented, only CPU usage was monitored and a suitable step duration threshold specified (sections 5.6.5, 5.7.3). Every step the RM obtained the current CPU usage for each process on the node and if the total consumption for all entities exceeded the threshold, the most expensive entity was volunteered for migration. The processor usage for the RM and the UM was not included in the total to simplify the charts. Unlike the entities used in the previous tests which had a uniform workload, a random element was programmed into each entity which would trigger a gradual increase in CPU usage. After peaking, this consumption would diminish until the entity's original workload level had been reached. In all cases a total of 40 entities system-wide was used and the threshold was set to 65 ms (with the intent that 70 ms would not be reached), beyond which a migration is required. The MUM does not actively load balance in these tests which rely on the passive mechanism triggered by a threshold violation.

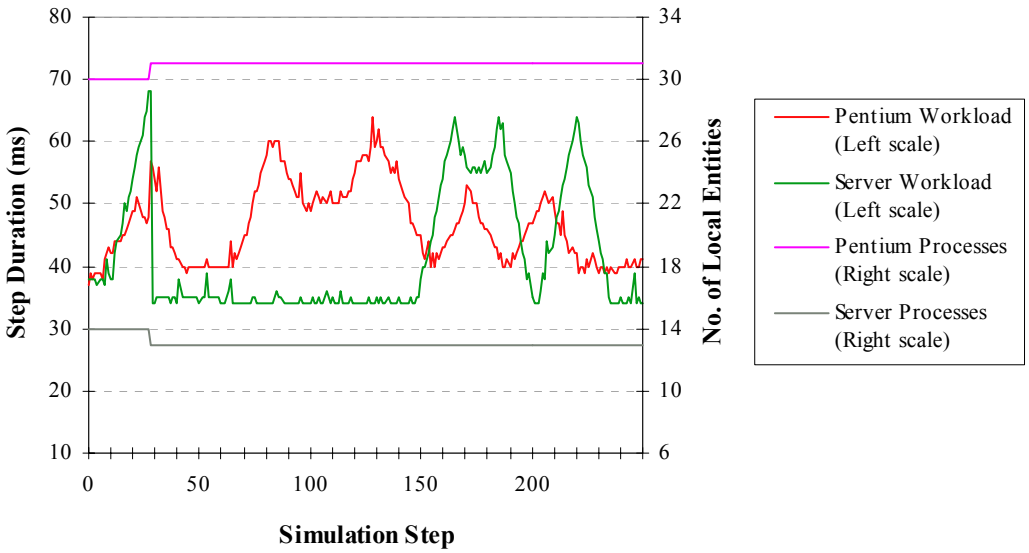


Figure 6.33 Single entity migration within a two node system (Pentium/Server).

Figure 6.33 shows two time series which represent the workloads of the two nodes in the test system and the number of entities present on each node (measured using the scale on the right). The peaks on Server are much higher since it has the slower CPU and at the 30th step exceeds the threshold resulting in a migration. The only place for the entity to go is the master node (Pentium) which already has an entity on the downward slope of a brief workload increase. Both nodes progress as other entities experience increases in workload. The double peaked feature at step 175 represents the product of the workload of two entities, one decreasing, the other increasing. After 250 steps, the nodes have gone from the same starting workload to one that differs by 6 ms. This does leave room for moving a few entities around to improve the load balance if the MUM was actively load balancing.

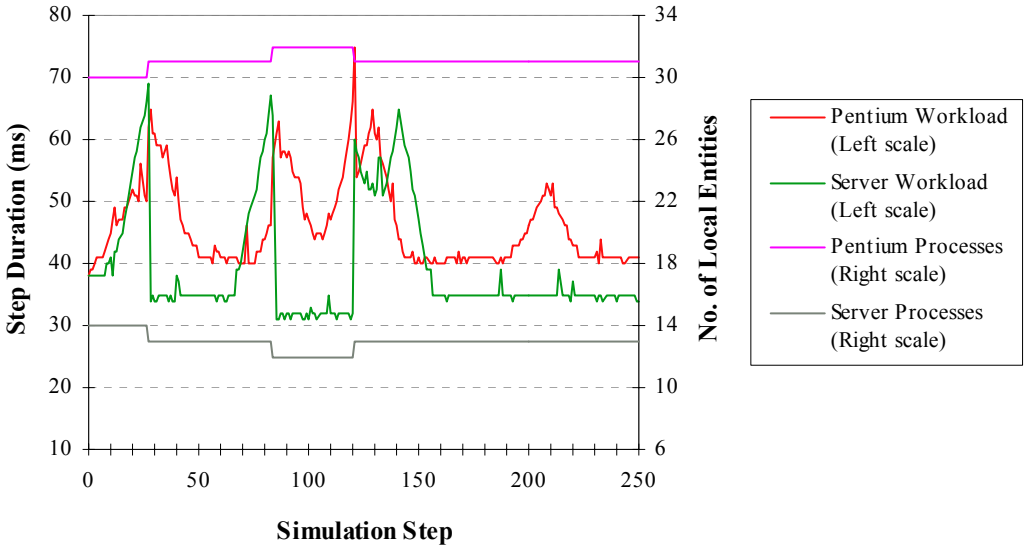


Figure 6.34 Multiple entity migration within a two node system (Pentium/Server).

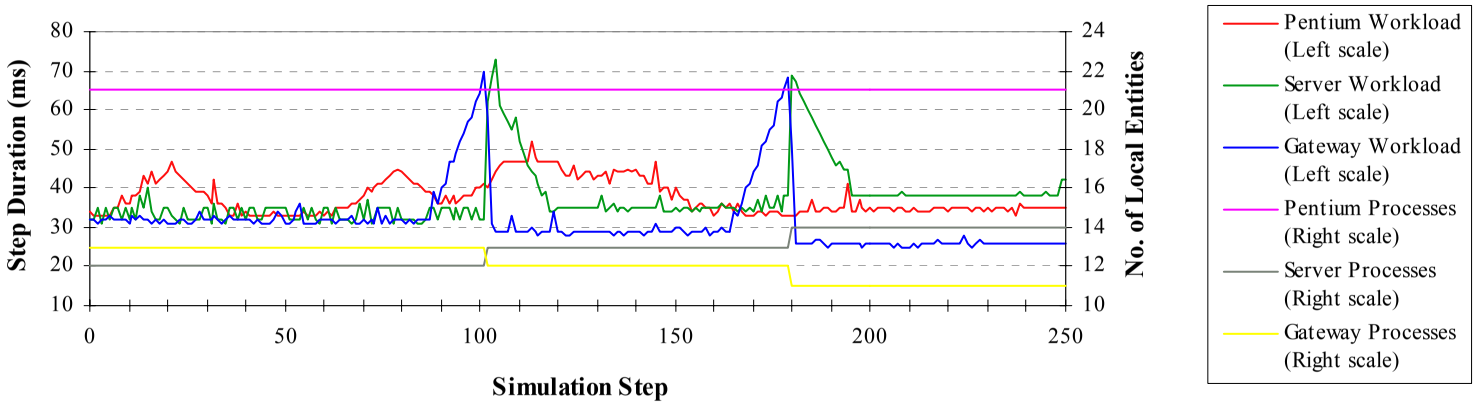
Another example of entity migration on two nodes is shown in Figure 6.34 where two entities on Server cause the threshold to be exceeded. This time an entity also pushes Pentium over the edge at step 120 and it is migrated to the slave node.

The charts in Figure 6.35 show entity migrations occurring in a three node system. In Figure 6.35a two migrations are required from Gateway and in both cases the chosen target node is Server. Each time the target node's CPU also exceeds the threshold, this is due to the fact that the state construction performed for each entity directly after migration is more expensive than the update function. For this reason, after a process has been created a four step period[6] is used to wait for the CPU consumption to settle down to normal levels. If this hysteresis period was not in force then the target node would immediately reject the new entity; an action which could be repeated any number of times resulting in the entity bouncing between nodes and thus destroying system performance. The second migration shown in Figure 6.35b is from Pentium to Gateway which is clearly a mistake on the part of the load-balancing algorithm. Even after the resting period, CPU consumption is far too high and the entity is migrated to Server.

The workload patterns in these tests were contrived but clearly demonstrate the migration mechanism. It is also clear that more comprehensive information must be used to determine the target node in order to avoid misallocations, i.e. a full RM implementation is needed (section 5.7). If some allowance was made for a short burst of CPU when the new entity is constructing, it would be possible to remove the current four step settling period.

[6] Discovered by empirical means.

a)



b)

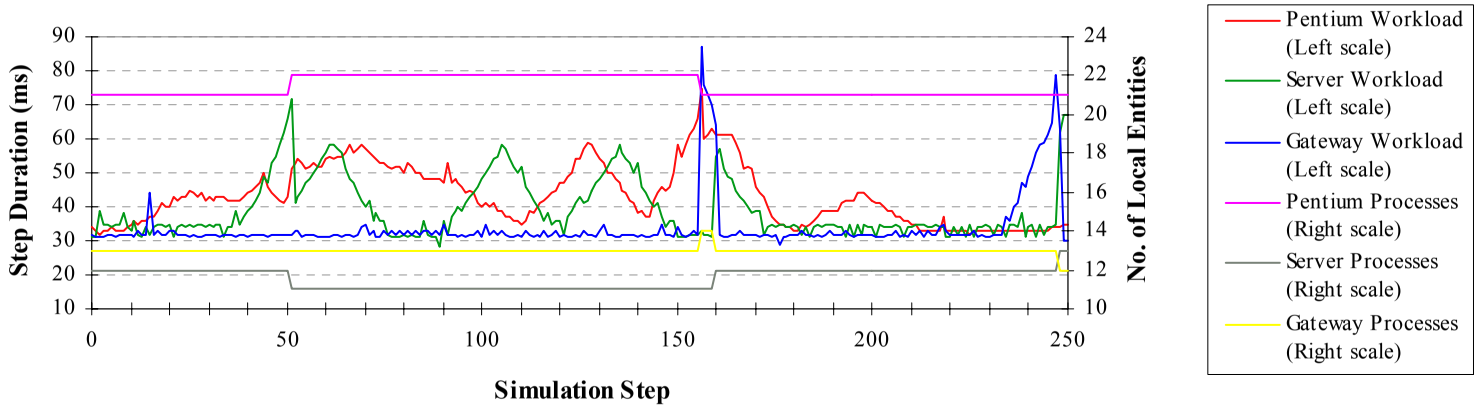


Figure 6.35 Multiple entity migration within a three node configuration: a) migration between slaves only; b) migration between all nodes.

6.5.5 Process Activity

Currently the time between starting a process and it reaching alive status is more in the region of hundreds of milliseconds rather than a few milliseconds. The most intensive part of this time is the creation of the main component process and the mailer. Following this the allocation of the UPID must take place and then the process' internal initialisation which can vary depending on its purpose, i.e. manager or entity. The actual creation time for a process also depends on the number of other processes starting at the same time. For example, when initiating a simulation with 40 entities, all entities and managers may not reach an active state until 30 seconds after the UM was started. Creation of a process on a slave node by the MUM is further confounded by the communications latency.

From the UM's perspective, the termination of an entity is quicker because the actual process termination is faster and the administration overheads are comparable with creation, e.g. informing managers of an entity's death.

6.5.5.1 Benchmark Entity

An integral part of the entity process creation/termination is the execution of that entity's specialised construct and destruct functions respectively. The duration of these functions in the entity that was used in all but the migration benchmarks is shown in Figure 6.36. Execution of the entity's destruction routine takes longer than construction because constructing a UML component generally takes less time than destructing it (section 6.3.5). This does not hold true for Reality in this case, probably because the unoptimised code for construction is actually slower than the operations needed to free memory.

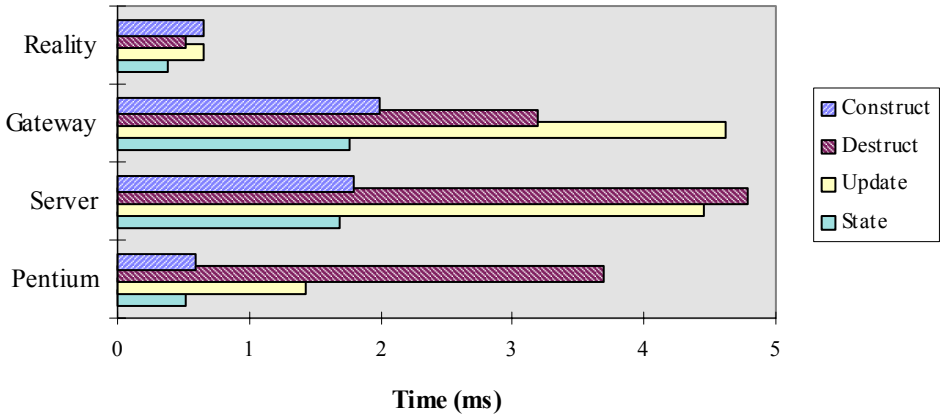


Figure 6.36 Construct/Update/Destruct times for the entity used during benchmarking.

Approximately 38% of the time that an entity uses when updating is spent sending the state data to the UM. The remainder of the time is used by the invocation of the entity's update function prior to the state transmission which, in this case, merely toggles the flag to indicate that the component has been modified (although its value is not actually changed).

Measuring the performance of an entity with the current prototype is somewhat problematic since the duration of any given task is totally dependent upon scheduling. As the number of entities grows the variances in measured duration become more profound; despite getting the same amount of CPU each time. The best way to measure an entity's performance, therefore, is to restrict the simulation to one entity such that it is unlikely to be interrupted during measurement. Figure 6.37 shows that the entity used in the benchmarking is idle for ~79% of the simulation step under QNX, the update taking at the most a few milliseconds. Again, due to poor TCP/IP performance Reality is already idling at 99% along with all the other system components. If the number of entities in the simulation was increased we can expect to see an increase in both the time it takes to send a state update and the idle time of the process.

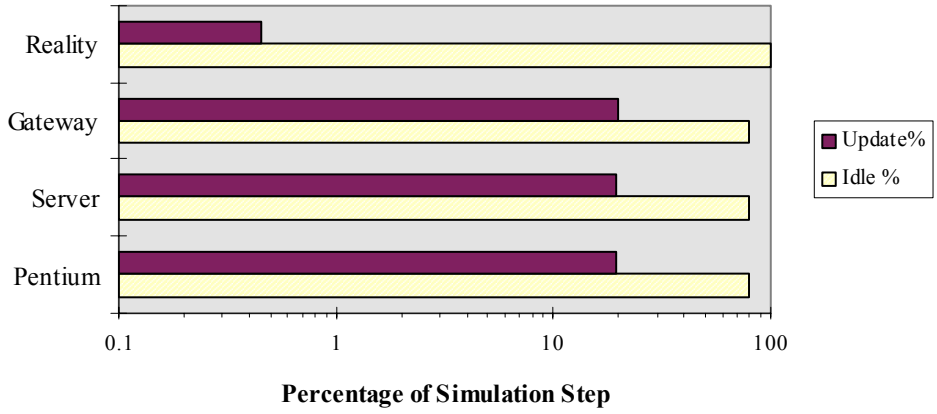


Figure 6.37 Average task breakdown for a single entity.

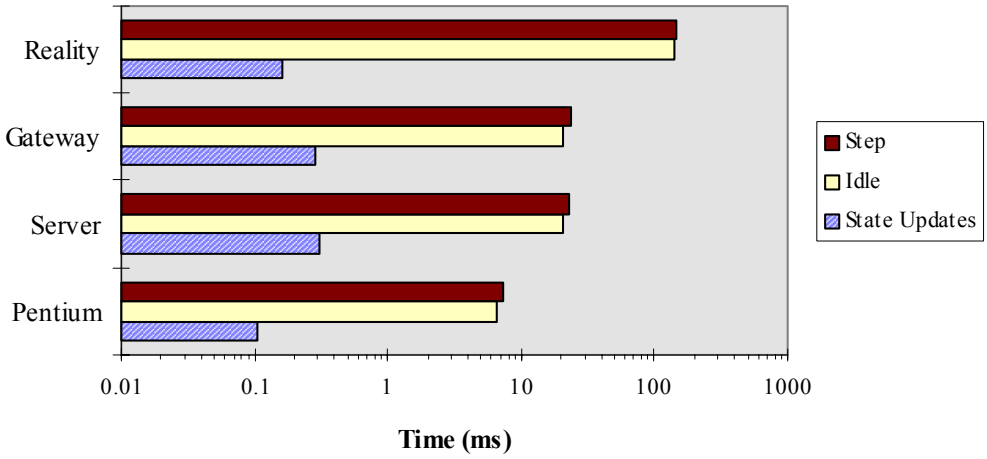


Figure 6.38 Activity breakdown for the benchmark manager.

6.5.5.2 Benchmark Manager

In the same way that most of the entity's life is spent idle, so is that of the manager used in the tests when there is only one entity in the simulation (Figure 6.38). The ratio of processing state updates to the total step duration will always be small, because a manager only performs its work at the end of the step when all entities have sent their updates. With just one entity

the manager is idle around 88% of the time under QNX, but an increase in the number of entities will increase the time spent processing updates, the step duration and the idle time.

6.5.5.3 Resource Manager

The most intensive activity that the current RM performs is obtaining the CPU usage of each process on its node. The basic overheads specified in Table 6.9 include the cost of monitoring itself and the UM; the costs of checking CPU usage for an entity or manager is also given. Usage of the PML mailer process is included in the calculated resource ratings. It is clear that this process is quite computationally expensive if performed every simulation step - as it was in the migration benchmarks. However, the current version does not attempt to perform any usage predictions that may be used to reduce the frequency with which this monitoring is required.

Overheads	Pentium	Server	Gateway
Basic (ms)	5.72	20.85	19.17
per Process (ms)	0.8	2.705	3.84

Table 6.9 Time penalties incurred when RM monitors CPU usage.

6.5.6 Simulation Execution Summary

This section has concentrated on the performance of the USS as a whole. Each of the test platforms, with the exception of Reality, were examined as single node systems. This provided a basis for evaluating performance when they were combined in two and three node systems. Although the same tests were performed on Reality and Gateway using TCP/IP, due to the problems with its use, each process spent 99% of their time waiting for messages. This issue is dealt with in section 6.6 which looks at improving the prototype's performance.

It was found that the most limiting factor in a distributed configuration was the network latency and that it had a substantial impact on performance. By carefully allocating processes to nodes, a three node configuration was shown to produce the same performance as an equivalent two node configuration. However, for the same number of entities this was still many times slower than simulating all the entities locally. This is not a common situation since the entities in question did no real work and used little memory. Given computationally more expensive or physically larger entities, distribution becomes a necessity rather than a luxury.

Often entities will consume different amounts of resources at different times which has the net result of producing a variable node workload. By migrating entities from one node to another, the available resources may be utilised to the maximum. To test the migration mechanism, multi-node systems were stressed with a number of entities, each with a variable computational workload. Finally, it was shown that it is possible to monitor resource usage and move entities in order to keep the system workload relatively evenly balanced.

6.6 Improving Performance

Based upon the knowledge gained from the analysis of the prototype presented in this chapter, it is clear that there are a number of improvements that can be made.

6.6.1 Message Elimination

One of the most limiting aspects of the current implementation is the use of point-to-point communications between processes, especially the UM and its entities. Figure 6.39 indicates the percentage performance increase that would be experienced if the update notification messages could be sent to all entities simultaneously rather than sequentially, i.e. an inter-process multicast. This was calculated by replacing the usual linear increase of time for the update task with the time taken to update a single entity; all other overheads were left untouched. The figure shows that a multicast method would produce greater performance benefits as more entities are added to the system. If the chart was extended by testing the method with more entities the performance increase would remain about 150%. The overall effect is not as dramatic with those configurations that transmit more state information. Also, the impact is diminished because each entity must still inform the UM that it has completed updating every simulation step.

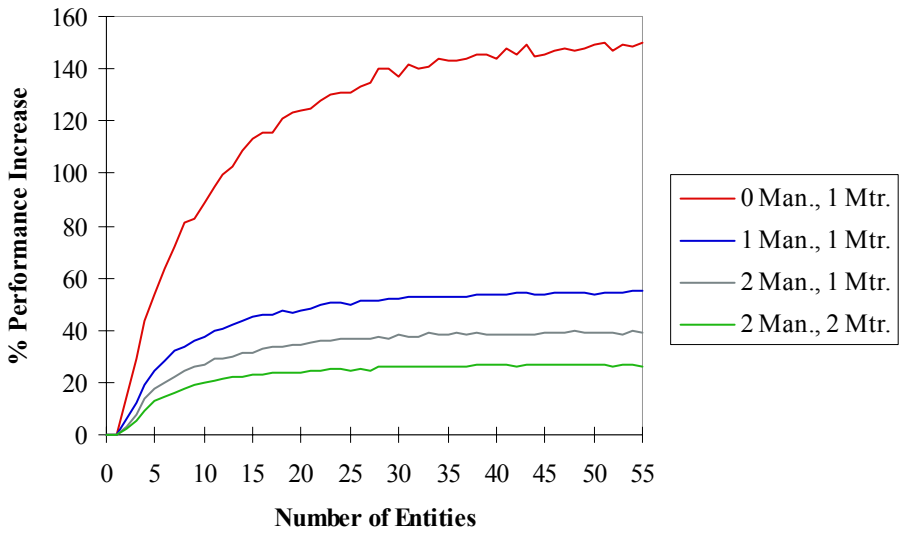


Figure 6.39 Potential performance increases with a multicast update (Pentium).

However, use of both the update notification and complete messages within a node is actually mimicking the behaviour of a deadline scheduler. When the scheduler triggers the entity to start processing, this may be taken as a cue to begin updating. After sending any state updates that were necessary, the entity would reach its deadline and this would indicate to the UM that the entity had finished updating. Therefore the actions that are currently performed explicitly with messages would be replicated implicitly by the nature of the scheduler. The greater the number of entities, the more time saved each step by eliminating the update complete message (Figure 6.40).

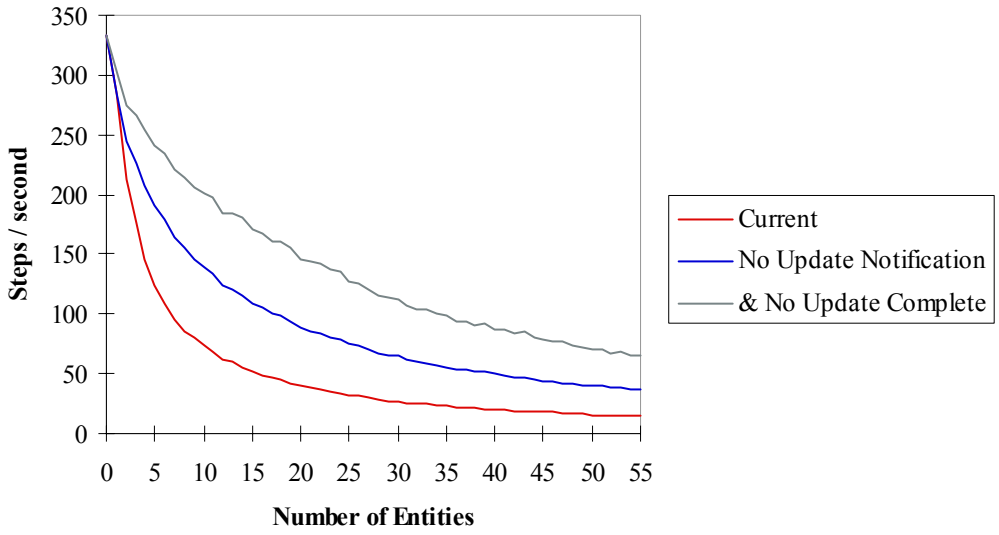


Figure 6.40 Increases in simulation rate when eliminating both update control messages: 0 Managers, 1 Monitor (Pentium).

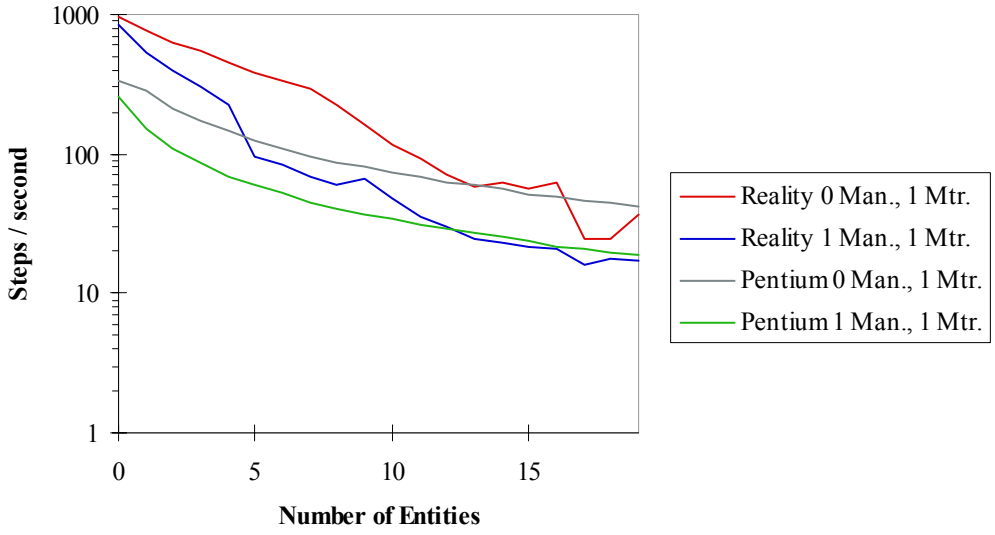


Figure 6.41 Comparison of estimated Reality performance with shared memory IPC and Pentium using QNX IPC.

6.6.2 Shared-Memory IPC

QNX IPC essentially copies a block of memory (the message) from one process' address space into another, therefore implementation of a shared memory based IPC protocol is unlikely to show much improvement. This is not the case when compared against the burden of using TCP/IP for local communications. The same simulation combinations executed on Gateway when using QNX IPC are a lot faster than when using TCP/IP on the same machine, e.g. 24 times for 1 manager/1 monitor, and 34 times for 2 managers/1 monitor. The idle rates of processes using TCP/IP on both Reality and Gateway are very similar (~99%) and a shared-memory IPC system will likely show similar performance to that of QNX IPC. Given this, it

is not unreasonable to use this speed-up factor as a rough indicator of the performance increase we could expect to see on Reality if shared-memory IPC was adopted. Figure 6.41 presents a comparison between the predicted performance and that of the fastest QNX node using the native IPC. Saying any more than that the two machines now present comparable performance would be unwise given the uncertainty of the estimation procedure used.

6.6.3 Multicast

Of course, TCP/IP is still the only available reliable method for communicating between heterogeneous machines on a network. This is also an area that could be optimised through the adoption of a reliable multicast protocol. A MUM could multicast update notification messages to its slaves and its use would also open up the possibility of state multicasts. The Single Connection Emulation sublayer presented by Talpade and Ammar (1995) is designed to sit between an existing reliable transport protocol and the network layer providing the unreliable multicast capabilities. The presented implementation used TCP as the transport protocol and IP as the network layer. The existing TCP API is utilised as usual but is supplemented by a direct interface to the SCE layer in order to control the multicast-specific variables of the multicast connections. One advantage of this approach is that it is possible to modify the semantics of multicast connections by changing the SCE without affecting the transport protocol. Unfortunately, because TCP is used, this solution also requires that prior to transmission a connection is established from the source of the multicast to the set of destination nodes. After transmission has concluded the connection must be closed. Consequently, in order to make use of this solution, the modifications to PML operation discussed in section 5.3.5.3 would have to be made. Nevertheless, of the solutions to providing a reliable multicast service that the author has seen, this seems like the most promising. In addition, should a more suitable reliable transport protocol come to light, SCE could be adapted for use with it.

The biggest savings that can be made are with the transmission of state information which is the most common and often the largest type of message that is sent. Each inter-process and inter-node communication pathway has a unique monitor ID associated with every component whose state is transmitted along it. This method works well for point-to-point links where the monitor ID is modified as the state is forwarded to all interested parties, but precludes the use of multicast in any form[7]. A possible solution to this problem would be to replace the monitor ID by the component's absolute name within the UML definition. The implications of this change would be an increase in message size (the absolute name could be potentially very long), and an increase in the amount of time needed to identify the component in each process' internal data structures prior to unpacking.

The adoption of multicast communications between machines would remove some of the burden from the master USS; it would also use less bandwidth. Consider a system with a master and two slave nodes: currently state information from a slave is sent to the master which forwards it to the other slave (if needed). With a multicast only one message would be required which would reach both nodes simultaneously. This cuts the required bandwidth by

[7] Multicast between processes on a node may be simulated through a shared-memory buffer that is monitored by all processes.

half and as message sizes and the number of slaves increases, so do the savings. In addition, it is possible that the component dependency list could be used to form a multicast group for those machines interested in its state updates. Although this would not reduce bandwidth consumption, it would ensure that any node not interested in the multicast did not waste time processing the message. Using a shared-memory emulation of multicasting, such gains as these could also be experienced by processes on the same node.

The application of this technique promises to yield significant performance increases but the computational cost of supporting it is uncertain. It is, however, an area worthy of further investigation.

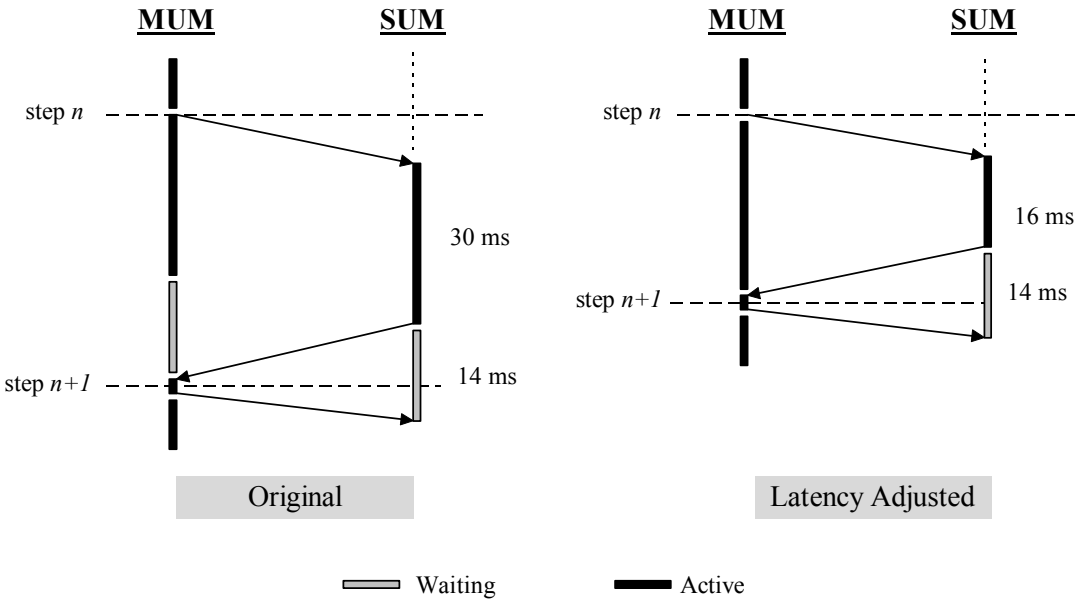


Figure 6.42 Accounting for message latency reduces simulation cycle duration.

6.6.4 Accounting for Latency

Section 6.5.2 presented a situation where the slave node spent a considerable amount of time waiting for the next simulation step to begin. Although it is not possible to eliminate the time the SUM spends waiting, its counterpart in the MUM may be removed if the slave's workload is reduced such that it finishes its work earlier. Figure 6.42 shows the current situation on the left hand side where the MUM (with an identical workload) has to wait for the SUM to respond before it begins the next simulation step. If the SUM's workload is reduced by 14 ms from 30 to 16 ms then the MUM no longer has to wait, thus increasing the simulation rate. In the case presented in Figure 6.28 this technique would effectively require the reduction of the SUM's (and the system's) workload by 5 entities. Thus, when the MUM is managing 31 entities, the SUM would be coordinating 4.

6.6.5 Increased Bandwidth

Increasing bandwidth would not obviate the need for the technique presented in the previous section, but it is the simplest way of improving performance. The results presented in this

chapter were based upon a dedicated Ethernet link with a theoretical speed limit of 10 Mbps. Recently, networking mediums such as FDDI and Fast Ethernet, capable of operating at 100 Mbps, have become widely available. Figure 6.43 shows the impact that using a 100 Mbps link between machines would have on messages sent between Pentium and Server. This prediction is based upon three assumptions: firstly, that the network throughput would experience a seven-fold improvement[8]; secondly, the bus can cope with the increase in required data transfer rate[9]; thirdly, that the same level of utilisation QNX currently achieves would be increased by the same degree (section 0). Whereas it took between 2 and 25 ms to send a message between nodes, this now occurs in 0.3 to 3.7 ms. The result is that latency is now lower than the PML overheads which now account for approximately 55-65% of the total send time.

The impact this would have on the master-slave benchmarks presented is difficult to estimate due to the unknown scheduling factor, but all messages sent were less than 1K in size. This would mean that a few milliseconds would be saved on each transmission. Considering the case presented in Figure 6.29 this would probably result in a latency reduction of around 23 ms (1 update complete + 1 update notify + 9 state updates). Currently all processes idle enough to cope with this decrease in transmission time but consideration of a more complex case would require further investigation.

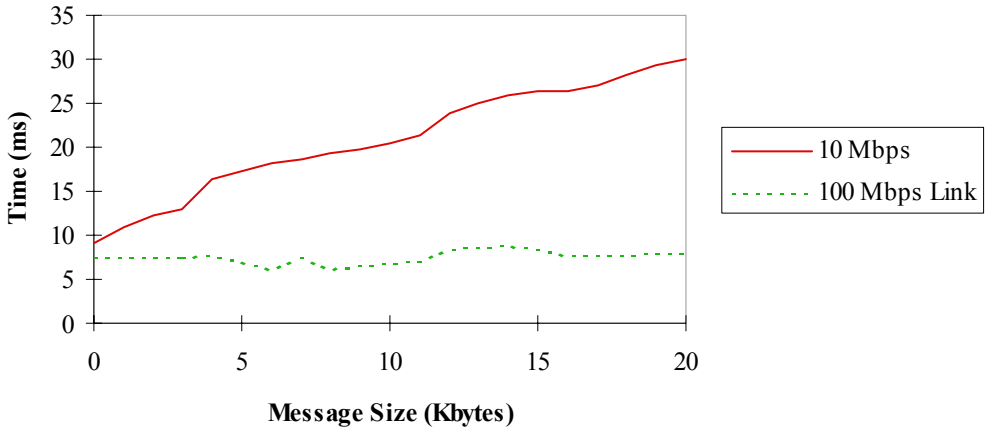


Figure 6.43 Pentium to Server transmission times when using the current Ethernet link and a predicted Fast Ethernet link.

6.7 Summary

This chapter has presented an evaluation of a prototype USS concentrating on the modeling language, the characteristics of the message passing systems and general simulation

[8] This assumption is based upon manufacturer's data that states a data transfer rate of 7.4 Mbytes/sec for 100BaseT Fast Ethernet as opposed to 0.9 Mbytes/sec for 10BaseT.

[9] An ISA bus cannot match the demands of a 100 Mbps network, whereas a PCI bus will.

performance. A number of points were made in the section summaries throughout this chapter but there are a few observations and aspects worth emphasising.

6.7.1 Living with TCP/IP

Although TCP provides a reliable connection, it uses an unreliable medium (IP). Positive acknowledgement is used to ensure that packets arrive at their destination - failure to do so results in retransmission. The greater the distance between the connection's two endpoints (and the more routers, etc.) the longer it will take to determine whether a packet has been successively received. The use of a hierarchy to connect nodes (and processes) in a USS and a network of USSs provides a more robust communications mechanism than requiring a single process to communicate with a server over some large distance. Should a link fail then this can be detected far quicker because the distance between nodes is far less. Resolution of this problem can be handled by the node that detected the problem or the sender can be informed and action taken accordingly.

This information also supplements the determination of whether the destination node is still alive. Although routers report when they cannot deliver any given message using the Internet Control Message Protocol (ICMP - Postel, 1981b), they may not be able to detect all such errors. The ability to detect errors is dependent upon the hardware protocol. For example, Ethernet does not acknowledge transmission of packets meaning that a node can be disconnected without affecting the rest of the network. Unfortunately, this also means that with Ethernet it is not possible to detect power failure, etc.

All of the message size tests are dependent upon the Maximum Transfer Unit (MTU) which may be different for all network media. For example, Ethernet has an MTU of ~1500 octets[10], whilst FDDI has an MTU of 4770 octets and ATM uses 9180 octets[11] (Laubach, 1994). If a message is transmitted greater than the MTU in size then it is fragmented. This fragmentation and corresponding reassembly at the destination inevitably incurs some overhead. In addition, the more fragments the greater the chance that one will be lost and the message will be discarded.

TCP/IP has its faults but it is the backbone protocol of the Internet and, in one form or another, it is here to stay. It would seem prudent, therefore, to find some way of working with it.

The Hyper Text Transfer Protocol (HTTP) is used by the World Wide Web (WWW) to retrieve distributed objects. HTTP uses TCP as the transport layer enabling WWW servers and clients to communicate. Every time a request is sent from client to server a connection is established, e.g. loading a new page, new icon/in-line image, etc. As the author discovered when evaluating the performance of the PML, this is an inefficient way of using TCP which is designed to handle data transfer over connections established for long periods of time, e.g. file transfer. Spero (1996) presents a detailed summary of the problems with the way in which

---

[10] Some implementations vary from the Ethernet specification.

[11] ATM could handle 64K octets but has been limited to 9180 so that it is compatible with the older Switched Megabit Data Service (SMDS) technology (Piscitello and Lawrence, 1991).

TCP is used by HTTP, including TCP's TIME-WAIT state (section 6.4.4). A proposed solution to these problems is the incorporation of a simple multiplexing protocol to be used with HTTP, enabling multiple requests to be dealt with on a single connection (Gettys, 1996).

The advantages of shared memory IPC over TCP as a local IPC mechanism have already been discussed. However, permanent connections could be established between key components on each node using TCP/IP, for example MUM to all SUMs and between systems, which are the links that need to be optimised the most. The price for this change is increased complexity within the PML which now has to handle two different types of connection. Nevertheless, applying TCP/IP in this manner would be more efficient than the way it is used now and is the equivalent solution to the multiplexing protocol mentioned earlier.

6.7.2 Resource Management

The amount of memory used by each program's execution image alone is larger than necessary because shared libraries could not be used. Currently each entity process under QNX requires approximately 256 Kbytes of memory (section 6.2.3); with shared libraries this could be reduced to the order of 6 Kbytes. Around 330 Kbytes of USS shared libraries would be shared amongst all processes in addition to ~55 Kbytes of system libraries.

The memory needed to store the UML definition and its instance data is quite large. Unfortunately this is the price that must be paid in order to effect modifications at run-time with as little disruption as possible. There is a relationship between definition structure and the amount of resources that any given operation consumes. This type of information could be used by the interpreter to predict how long an operation will take or how much memory a definition will require. Accurately predicting resource consumption aids the RM in its work.

The total simulation workload is unlikely to remain the same throughout the simulation and local fluctuations are to be expected. The migration mechanism presented is currently in a primitive state but adequately demonstrates the benefits such a technique can have on system loading. More resource utilisation information is needed so that better decisions can be made about a node's loading and predictions of an entity's workload.

6.7.3 Scaleability

A detailed analysis of the PML proved that communicating between machines is many times slower than between processes on the same node. An unexpected problem with TCP/IP was encountered which, combined with its use by the PML, made using it as a local IPC mechanism impractical. Examination of the PML provided a basis for evaluating system performance as a whole and also enabled predictions to be made by modifying key variables such as bandwidth.

A number of simulations were run on single node systems, each one using a different number of managers and an increasing number of entities. The results showed that, in general, more time is spent processing state information that any other type of data. Also, the UM spends a considerable amount of time idle waiting for other processes to complete their work. This analysis of single node systems provided a basis for evaluating multi-node configurations.

6.7.3.1 Standalone USS Performance

Performance in a USS is dictated by a number of factors (in no particular order):

1. Number of special managers.
2. Location of the special managers/entity processes.
3. Number of entity processes possessing monitored state.
4. Size of the state information monitored by those managers.
5. Transmission frequency of the monitored state.
6. Number of USNs in the USS.
7. Latency/bandwidth of the connections between the nodes.

The more managers that monitor any given part of the whole VE's state, the more state information that must be sent between processes, i.e. the more bandwidth consumed. The best case is if the manager in question is interested in just the state held by entities on its local node: where the available bandwidth is highest and the latency lowest. The more common case is when a manager is interested in state held by entities that are spread on many nodes within the system. In this case the size of the state information that must be sent to the manager(s) becomes even more important - the more state information or the smaller the link's bandwidth, the lower the performance. If the manager is on a slave node then an entity on another slave node will send its state update to its local UM, which forwards it to the MUM, then onto the manager's local UM and finally to the manager itself. If the manager is on the master node then this procedure takes one stage less. If the entity is on the master node and the manager is on a slave node then the procedure is also one step quicker.

The amount of state data sent is dependent upon the frequency of changes to that state made by each entity. It is not possible to calculate in advance what this frequency will be since it is semantic specific. A well designed manager will monitor information that changes on a periodic basis and make use of constraint functions. These can be used to filter, at source, the state data before it is sent to the manager consuming valuable bandwidth and processing time.

Although the number of nodes in a system and the speed of their communications links plays an important part in performance from a state management perspective, they are also relevant when considering synchronisation. At the beginning of each simulation step the master node synchronises all the slave nodes through an exchange of messages. Using unicast, there is a linear relationship between the time taken to perform this procedure and the number of nodes in the system. Again, this could be partially rectified by replacing the initial master-to-slave synchronisation control unicast with a multicast.

As the reader can see from this list of confounding factors, it is difficult to build a clear picture as to exactly how performance will scale as more managers and/or entities are added to the system. What is clear from the results presented in this chapter, is that performance will fall sharply initially, and then gradually asymptote as more processes are added. However, this could be dramatically scaled down if multicast was used to send state updates to interested managers (Figure 6.39). Not only would bandwidth be saved but the burden on the MUM as a router would be reduced significantly, thus removing what would become a major bottleneck in the system as state information flow increases.

6.7.3.2 Networked USS Performance

Performance between USSs is also dictated in a number of ways:

1. Total number of users across all systems.
2. Type of information sent between systems.
3. Method used by the user's shadow process to approximate behaviour.
4. Number of networked systems executing same simulation.

The amount of traffic on the inter-system links is mainly due to two related factors. Firstly, the more users participating in a given simulation, the more user-specific data that must be sent to all systems executing that simulation. Secondly, the amount of information is dependent upon the type of data being transmitted. For example, low-order information such as position and velocity will be sent almost continuously whereas high-order information indicating changes in behaviour will be sent less frequently. It would seem therefore that the latter would guarantee better performance. Unfortunately this requires a more complex shadow process to interpret the information and do something sensible with it. Balancing the amount of data and the amount of computation a shadow requires to process it is the key to good performance.

With the current hierarchical structuring of systems performance, there will be a non-linear degradation in performance as the number of systems increases. Not only will the effort expended by the MUM in each system be increased due to routing but the time taken to ensure every system gets the transmitted message will also grow. Again, multicast will relieve this problem, allowing a single transmission to reach each system running the simulation. Using this technique, performance should be mainly affected by the number of users in the system, not the transport mechanism.

6.7.4 Distribution at a Price

When the author started this work, distribution seemed like the answer to all the problems regarding limited resources and multi-user interaction. It is now clear that there is a distinct price to pay for distribution and it should only be considered if the advantages outweigh the disadvantages.

Communications latency is presently the largest factor responsible for inhibiting progress of a distributed simulation. In simple simulations there is little to be gained by spreading the load throughout a network of machines because more time will be spent communicating than actually performing simulation work. Only when the simulations become more expensive is this cost offset enough to prove beneficial. The advantages of distribution include the possibility of multiple users. It should be noted, however, that the desire to include more users in a VE may well degrade performance due to the problem just mentioned. At the other end of the scale, if the presence of many users generates too much state information flow, then it will not matter how much computation there is to perform.

6.7.5 Conclusions

The prototype is not perfect and several enhancements that would improve performance have already been discussed. Some, such as the deadline scheduler, require more specialised operating systems whilst others, such as multicast, need a combination of hardware and software protocols that is not currently readily available. Fortunately, a shared memory IPC mechanism could be implemented now, as could the technique used to account for transmission latency.

The balance between CPU performance, bus speed, memory capacity and network bandwidth (amongst others) is an important one; a well configured system will take all of these into account. For example, if only network bandwidth is increased then eventually there will come a time when the bus may become the bottleneck, or the CPU is incapable of processing data fast enough for transmission. The relationship between these factors is influenced by the software system. Unless analysis of the type presented in this chapter is performed, i.e. at the component and system levels, systems engineers will not be able to deliver the technology capable of supporting distributed VE systems.