Efficient Access of Remote Resources in Embedded Networked Computer Systems

Paul S. Usher and Neil C. Audsley

Real-Time Systems Research Group
Department of Computer Science, University of York, York YO10 5DD, UK
usher@cs.york.ac.uk

Abstract. Fixed networks of limited resource heterogeneous computers need to allow applications to access remote devices in order to overcome any local resource deficiencies. Current operating systems would use a file system, network stack and middleware to implement such access but the volume of functionality involved can be a barrier to performance. This paper examines the 2.4 series Linux kernel to show that networked operating systems lack flexibility and performance in this environment. It also presents a low level approach that can reduce the overheads incurred and improve performance when remote devices are accessed.

1 Introduction

This paper is concerned with fixed networks of heterogeneous single processor embedded and ubiquitous computing devices operating in close proximity and in networks that are largely static in nature. This means that only one type of communication medium may ever be used and the physical address of a resource may often be known in advance. The primary issue is therefore how to structure the operating system (OS) so that it is able to quickly and efficiently facility access to any remote devices required by the application in order to overcome local inadequacies. Such problems can in specific circumstances be addressed using additional hardware, both SMP and NUMA architectures are evidence of this approach but such designs are not in the context of this paper.

This paper analyses the 2.4 series of Linux kernel to illustrate the performance of a typical networked OS. It then shows how a low level approach that embeds a simple file based protocol directly into an Ethernet frame can make better use of the hardwares characteristics improving both performance and flexibility.

The remainder of this paper is structured as follows; section 2 examines the architecture of the Linux kernel in order show why a networked OS is not ideally suited to this tightly constrained environment. Section 3 supports this with a performance analysis of a typical GNU Linux based OS. Finally section 4 outlines how the large packet sizes of IEEE 802.3 and IEEE 802.11 can be combined with the PSE51 embedded systems profile of IEEE 1003.13 to remove much of the complexity involved with accessing remote devices [1–3].

2 Background: A Networked OS

A Networked OS such as Linux uses a stack of software for process control and communication, another for resource access (VFS) and another for remote communication (network stack) [4, 5]. Layers of functionality are then used within these stacks to aid flexibility and further simplify implementation. Accessing a remote device using this model requires the client machine to have some functionality (a proxy) connecting the file system and the network stack so that the application can access a remote device in the same manner as a local device. Alternatively if a separate interface is used the functionality must implement the operations via the existing network stack. The server also requires similar functionality in order to access the device on behalf of the remote application. Consequently a considerable amount of functionality is involved as the flow of control passes up and down the file system and network stacks on both the client and server machines, see Figure 1.

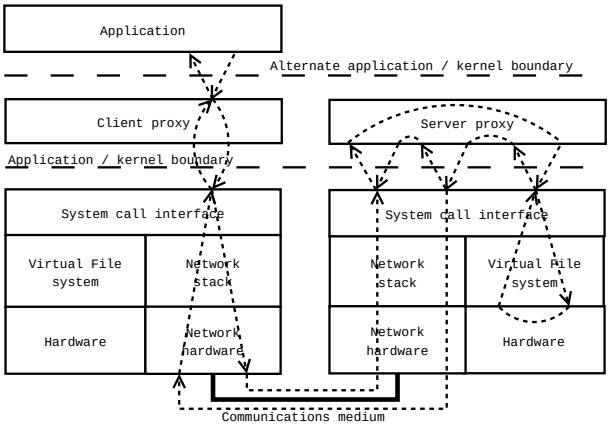

Fig. 1. Control flow when accessing remote resource access

To address the performance issues of this model the proxy processes can be moved into the address space of the kernel in order to reduce the amount of copying and the number of context switches, but this does not reduce the volume of code involved in navigating the VFS or network stack [4, 5]. The network stack also does not allow the characteristics of the delivery device to affect the operation of the layers above it [4, 5]. Consequently local socket based communication is likely to be adversely affected as the fragmentation and redelivery functionality cannot be removed when the delivery mechanism is reliable (memory). It therefore seems clear that networked OSs are not ideally suited to constrained environments where flexible and efficient access is required to both local and remote resources (devices, files or applications).

3 Performance Analysis of a Linux OS

To demonstrate the performance of a typical networked OS this section uses the Lmbench suite of benchmarks (version 2.04) on an isolated and directly connected 10Mb two node network of otherwise idle Slackware Linux based computers [6]. The specification of the test machines is outlined in Table 1.

Table 1. Test machine specification

	Darkstar	Cheetah
CPU	AMD K6-2 350MHz	Intel Pentum II-MMX 266MHz
Memory	128MB	64MB
Hard disk	WDC AC28400R	WDC AC34000L
Network Card	3Com 3c590 10 BaseT	3Com 3c905C-TX/TX-M
OS	Slackware 9.0.0	Slackware 10.1.0
Kernel Version	2.4.22	2.4.29

The bandwidth of a network medium will govern the performance and characteristics of any networked system but its importance escalates as more advanced functionality is added. The addition of remote device access capabilities to a networked system begins to make it more like a distributed system, as a result the performance of the network medium becomes critical to the successful operation of the system as more and more applications become increasingly addicted to remote resources.

The availability of necessary quantities of network bandwidth clearly governs how much work a system can get done, however, it is also important not to achieve this at the expense of latency. Both figures will ultimately be dominated by the performance characteristics of the network medium and associated hardware. Although the sheer volume of functionality involved in the file system, network stack and proxy components of the OS is likely to have a limiting affect on the performance of an application, even if this only occurs at high levels of load.

3.1 Bandwidth results

The bandwidth benchmark transmits 3MB's of data between two processes in 48KB steps and then returns it once the data has been received. This is performed primarily between processes on the same machine since the majority of inter process communication is local, although processes on different machines are used if a particular mechanism supports remote communication, see Table 2.

A Pipe is a very simple communication mechanism supporting one way local communication and it is not surprising that it is the best performer. In contrast a UNIX socket provides two way local communication and achieves only 60-75% of a Pipes throughput. TCP sockets differ from their UNIX counterparts by

Table 2. Communication bandwidth (MB/s)

Communications Mechanism	Local		Remote	
	Darkstar	Cheetah	Darkstar	Cheetah
Pipe	82.8	143.0	-	-
UNIX Sockets	61.9	85.4	-	-
TCP Sockets	43.8	68.8	1.05	1.03

supporting remote communication across heterogeneous and unreliable communications mediums and when they are used in this fashion it is not surprising that they perform poorly in comparison to either UNIX sockets or pipes. However, they perform equally poorly for local communication (50% of Pipe performance) as the upper layers of the IPv4 stack remain unchanged even though the majority of their functionality is not required in the local environment. The effect of this performance loss should not be underestimated as it may not be possible to choose in advance an alternate communication mechanism in order to statically optimize performance. It might therefore be beneficial if the kernel was able to to optimise performance wherever possible, possibly by passing a UNIX socket of as a TCP socket.

3.2 Latency results

The latency benchmark uses a 1 byte "hot-potato" token that allows the resulting TCP or UDP message to fit into the minimum Ethernet frame (46 bytes of payload). It also tests Sun's RPC mechanism when used with both TCP and UDP and this makes it possible to better estimate the overheads incurred by a proxy process in a networked or distributed environment. Such functionality acts as the glue that binds the network stack and file system models together and an estimate of its performance therefore gives a more realistic view of an applications performance when remote devices are accessed, see Table 3.

Table 3. Communications latencies (μs)

Communications Mechanism	Local		Remote	
	Darkstar	Cheetah	Darkstar	Cheetah
Pipe	24.4	13.7	-	-
UNIX Sockets	52.2	25.6	-	-
UDP Sockets	77.6	55.2	231.7	235.5
TCP Sockets	107.8	88.0	279.1	280.4
Sun RPC over UDP	180.7	150.7	325.3	328.6
Sun RPC over TCP	230.7	204.8	425.9	434.3
TCP connection	416.0	340.0	451.3	476.3

Distributed systems require a suitable balance between bandwidth and latency. It is therefore concerning that TCP sockets incur 4-6 times the latency of

a pipe when used in a local environment, whilst UNIX and UDP sockets incur 2-4 times the latency in the same situation.

When considering the additional overheads involved in accessing remote devices it is important not to forget that a TCP connection must be established prior to its use and these results suggest that this takes around 340-470μs. This coupled with the costs of an RPC call over the same mechanism would seem to rule out the use of TCP for client/server style connections in any networked environment where the connection is not established for a considerable period of time.

The overheads for the RPC mechanism are relatively constant regardless of both the communication mechanism (UDP or TCP) and the location of the client (local or remote). For UDP based communication this overhead is around 95μs, whilst TCP communication sees this increase slightly to 120-150μs, See Table 4.

Table 4. RPC overhead (μs)

Communications mechanism	Local		Remote	
	Darkstar	Cheetah	Darkstar	Cheetah
Sun RPC over UDP	103.1	95.5	93.6	93.1
Sun RPC over TCP	122.9	116.8	144.6	154.3

4 Reducing Overheads

This section examines whether an OS whose architecture directly targets the need to access remote devices might reduce overheads, improve performance and still achieve sufficient flexibility.

Communications mediums such as IEEE 802.3 and 802.11 allow computers to deliver well over 1KB of data in an error free manner because of the capacity of the packet and the use of a 32bit CRC [1, 2]. If the majority of interactions could be made to fit into a single packet additional reliable delivery functionality may never be needed. In addition the embedded systems profile of IEEE 1003.13 (PSE51 of POSIX.13) indicates that a traditional file system is unnecessary in such systems. Instead sufficient flexibility is achieved by interfacing devices directly to the close(), open(), read() and write() functions, thus negating the need for the majority of functionality associated with a file system [3]. It therefore seems worthwhile to examine whether the relatively large packet sizes supported by these mediums can be utilised to directly encapsulate sufficient information to allow one OS to send file system requests directly to another OS without the use of either a VFS or the network stack. This reduction in the systems footprint would also allow it to be utilised in more restricted environments in addition to reducing the latency of any remote device access. The architecture of the resulting system is illustrated in Figure 2.

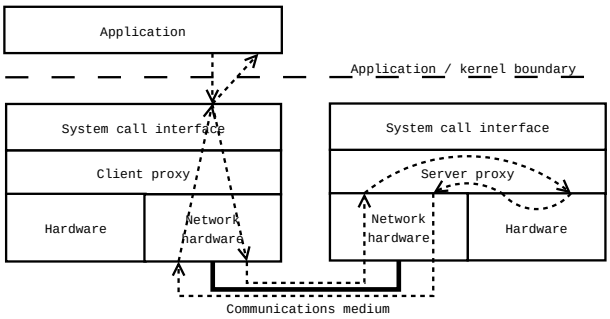

Fig. 2. Control flow when accessing remote resource access

4.1 Format of an Ethernet based file protocol

The following fragments of C code outline the format of the Ethernet frames when the parameters for the file operations are embedded directly into the data payload. This starts by outlining the format of an Ethernet header, and the types of messages supported by the protocol.

```
#include<stdint.h>

/* Declare an Ethernet header */
struct ethhdr {
        uint8_t dst[6];
        uint8_t src[6];
        uint16_t type;
};
/* Declare a file protocol instruction */
struct fp_inst {
        struct ethhdr header;
        uint16_t opcode; /* Instruction type */
        uint32_t tag; /* Instruction number */
};
/* Types of instruction used in the file protocol */
enum fp_opcode {
        FP_CLOSE,
        FP_OPEN,
        FP_READ,
        FP_REPLY,
        FP_WRITE
};
```

The "opcode" field uniquely identifies the type of instruction contained in the Ethernet frame in addition to aligning all of the following data onto a 32bit boundary in order to maximise performance. The client allocates the following

"tag" field in order to uniquely identify the request and to allow it to match a reply to the appropriate request. This kind of approach is used with some success in both 9P and Styx as it allows the server to identify incoming requests that have been repeatedly made by the client in order to overcome the unreliability of a networked system. The following code illustrates how the parameters for the various functions are encoded.

```
/* Declare a close instruction */
struct fp_close {
        struct fp_inst type;
        uint32_t fd;
};
/* Declare an open instruction */
struct fp_open {
        struct fp_inst type;
        uint32_t flags;
        uint32_t mode;
        uint8_t filename[];
};
/* Declare a read instruction */
struct fp_read {
        struct fp_inst type;
        uint32_t fd;
        uint32_t len;
};
/* Declare a write instruction */
struct fp_write {
        struct fp_inst type;
        uint32_t fd;
        uint32_t datalen;
        uint8_t data[];
};
```

The successful operation of a close(), open(), read() or write() function call may result in some data being returned to the caller and potentially some error code. In addition to this the read() needs to return some data to the client. The following structure could be used to represent this information.

```
/* Declare a reply instruction */
struct fp_reply {
        struct fp_inst type;
        uint32_t result; /* Return value */
        uint32_t error; /* Error code */
        uint32_t datalen; /* Length of returned data */
        uint8_t data[]; /* Returned data */
};
```

4.2 Payload utilisation

The file protocol outlined here provides a mechanism for efficiently allowing an application to access a device connected to a remote computer. Since the computers are all on the same network and the file protocol does not support messages bigger than a single frame it is possible to dramatically reduce the size of the headers required, see Table 5.

Table 5. Per layer comparison of protocol overheads in bytes

OSI	Protocol		
Layer	UDP	TCP	FP
Network	20	20	6
Transport	8	20	0
Total	28	40	6

The supported file operations require little data so they all fit comfortable into even the smallest Ethernet packet. This is particularly beneficial since it allows the maximum amount of data to be carried by those messages that also support a dynamic data portion. An open request for example supplies a file name, the length of which is only known at run time. Similarly it is not possible to know how much data will be written to a file, or how much may be returned from a read operation. Since all of this data must be contained in a file protocol message and these cannot span multiple Ethernet frames it is important that the dynamic portion of these messages is as large as possible so as not to reduce flexibility, see Table 6.

Table 6. Maximum size of dynamic data portion in bytes

Ethernet	Message		
payload	open	reply	write
46	32	28	32
1500	1486	1482	1486

The file name passed to open() can therefore be between 31 and 1485 bytes in length, since the last byte must be null in order to terminate the string. Whilst a single Ethernet frame is able to incorporate in excess of 1400 bytes for both the read and write operations. Transferring more data than this would necessitate breaking the larger operation up into multiple smaller requests, in addition to some support from the protocol for atomic actions so that either the whole request succeeds or it fails.

4.3 Operation of the server proxy

The results obtained from the testing of the Linux kernel made it quite clear that there are significant overheads involved with the packaging and un-packaging of data prior to it being sent to the server. The approach adopted here reduces these by limiting the amount of additional data that the protocol needs as well as only supporting a very small set of operations. This minimalist approach allows the operation of the server to be simplified. Handling of each type of incoming message (reply is outgoing) is offloaded to a function that is dedicated to the purpose. Dispatching is then a simple matter of checking that the opcode field is valid before using it as an index into an array of function pointers. Of the operations undertaken by the server the most expensive is read since it requires the allocation of sufficient memory to hold the data that is read from the file prior to it being sent back to the client.

The initial implementation of the protocol communicates with the remote machine via a packet socket that has been bound to a specific network connection (typically eth0). This allows both low level access to raw Ethernet packets and simplifies the implementation process, as well as allowing its performance to be analysed on the same two node Slackware Linux based system as has been used for the UDP/TCP analysis. Further development work is underway to integrate this functionality at a lower level within the kernel in order to further reduce any overheads. As a result the performance of both the client and particularly the server is likely to improve in the future.

4.4 Performance Analysis of Initial Implementation

The performance of this initial implementation has been measured through the use a benchmarking application that runs on one of the Slackware Linux machines whilst accessing files on the other via the file protocol implementation. The latencies are therefore those typically experienced by the application, see Table 7.

Table 7. Remote file operation performance

	Latency (μs)
close()	236
open()	242
read()	1535
write()	1695

The use of minimum sized Ethernet frames in both directions ensures that close() and open() perform better than read() and write(), which (in this case) utilise a full packet in one direction in order to maximise the amount of data transferred (see Table 6). The parameters supplied to these operations can have a

significant affect on their performance, open() in particular requires an additional 440μs when the truncate flag is used.

Given that this is an initial implementation these performance figures compare quite favourable with the 325-440μs required to transmit a single byte of data via Sun's RPC mechanism on UDP or TCP, especially as this figure does not account for any overheads incurred when the server process accesses the local files on behalf of the remote application.

5 Conclusions

It has been shown that access to remote devices requires some form of proxy and that current designs incur significant overhead both in terms of establishing a reliable connection, and in marshaling the data. It therefore seems unlikely that such software designs will achieve acceptable performance in embedded systems without the support of additional hardware resources. Although small network stacks are undoubtedly available it is dubious whether they provide any practical benefit in this context since the primary barrier seems to be performance rather than size. In addition it has been shown that a traditional networked OS provides unnecessary functionality in some areas and insufficient support in others. It has also been shown that there may be a potential size and performance benefit if the architecture of the OS makes better use of the resources it has available to it. To demonstrate this fact we have provided a simple Ethernet based file protocol that facilitates efficient access to remote devices with sufficient flexibility to satisfy the file based functionality required of a POSIX.13 PSE51 compliant system.

References

1. IEEE: IEEE 802.3-2002: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access method and physical layer specifications. (2002)
2. IEEE: IEEE 802.11-1999: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. (1999)
3. IEEE: 1003.13-1998 IEEE Standard for Information Technology — Standardized Application Environment Profile (AEP) — POSIX® Realtime Application Support. (1998)
4. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel. Second edn. O'Reilly & Associates, Inc. (2002)
5. Rubini, A., Corbet, J.: Linux Device Drivers. Second edn. O'Reilly & Associates, Inc. (2001)
6. McVoy, L.W., Staelin, C.: lmbench: Portable tools for performance analysis. In: USENIX Annual Technical Conference. (1996) 279–294