Application-Compliant Networking on Embedded Systems

Stefan Beyer, Ken Mayes and Brian Warboys
Centre for Novel Computing
Department of Computer Science
University of Manchester
M13 9PL
United Kingdom
Email: {beyer,ken,brian}@cs.man.ac.uk

***Abstract*— Network protocol stacks are traditionally encapsulated within system software, forcing the application programmer to use general-purpose communication end-point abstractions. The application programmer is denied the flexibility of implementing application-specific performance improvements. Application-level networking provides the application programmer with the ability to tailor the protocol stack to the needs of the application. This is particularly useful in special-purpose systems, such as embedded networked appliances. This paper describes the design of an application-compliant TCP/IP implementation for the Arena run-time library operating system, which aims at separating mechanism from policy. The role of policy and mechanism in network protocols and their effects on networked embedded systems is investigated. The resulting system is optimised for embedded systems based on a multi-threaded single-application model. Experiments were carried out on an embedded system test platform and performance results are given.**

I. Introduction

Networked appliances communicate through network protocols. Internet appliances are networked appliances that communicate over the Internet, such as Internet radios, TVs that can function as web browsers or burglar alarms that use the Internet to contact the police. Such an appliance uses the TCP/IP [1] protocol suite as its underlying network protocols. This introduces a problem, because different appliances might have different demands on the underlying network protocols. Designing special-purpose protocols for different devices is impractical in the case of Internet appliances, since such an appliance must conform to the TCP/IP standards. An alternative solution is to give the programmer of the application running on the appliance control over the implementation of the network protocol stack. Therefore, it is logical to implement as much of a network protocol stack at the end-points of the communication; that is, in the networked application itself. Salzer, Reed and Clark identified this as a general design principle in the *end-to-end arguments* [2].

Special-purpose operating systems, such as those commonly used with embedded-system-based networked appliances, support application-compliant systems. Such a system allows the application to tailor resource management to its needs. To achieve this, it is useful to separate mechanism from policy.

A mechanism/policy architecture is a layered architecture. Each layer (n) combines mechanisms provided by the layer below (n-1) by applying policy. This policy is used to provide higher level mechanisms for the next level up (n+1).

The Arena operating system [3] is a special-purpose operating system based on a separation of mechanism and policy. Mechanism is provided as low-level primitives by the nanokernel, whereas policy is implemented in user-level libraries. Arena is described in more detail in section III.

The present research applies the separation of mechanism and policy to protocols in the context of an application-compliant implementation of TCP/IP for an embedded-system version of Arena.

Although the *end-to-end arguments* have influenced the design of the TCP/IP protocol suite, implementations of the TCP/IP suite are typically encapsulated in system software. Furthermore, implementations are typically hidden behind relatively high-level communication end-point APIs, such as the UNIX socket API [4]. This separation of protocol implementation and application does not allow application-compliant communication systems.

The remainder of this paper is organised as follows. Section II gives some background, describes related work and introduces the model used in this research. Section III describes the Arena operating system which underlies the present research. Next, an attempt is made to categorise network protocols into mechanism and policy (section IV), followed by a discussion of buffer management (section V). Section VI gives the results of some performance experiments. Section VII describes possible areas for further research. Finally, section VIII gives a summary.

II. Background

A. Classification of approaches

Methods of improving network performance and network flexibility of applications can be classified into four categories (This categorisation is derived from Ganger *et al.* [5].):

- **(a) different protocol stack architecture**
  It is possible to define new protocol stack architectures,

such as reducing the number of layers or avoiding a layered approach completely.
Although these approaches might improve performance, they very often maintain the separation of protocol implementation and application, thus restricting the application programmer from applying application-specific optimisations.

- **(b) different APIs**
Many systems try to solve the limitations of the general-purpose abstractions that encapsulate protocols by inventing new interfaces or by improving the performance of the existing ones.
These systems might not only improve performance, but also flexibility. However they also fail to pass control over protocol stacks to the application. Encapsulating API implementations in inaccessible system software, will still mean restrictions for the application.
- **(c) Application "steering" of protocols**
It is possible to let the application control the way in which the network protocols are composed in the lower layers of the system. Generally, such a system allows the application to install "extensions" in its lower layers.
All solutions based on kernel extensions however, are solutions that operate in the traditional context of separating protocol implementation from the application.
- **(d) Application-level Networking**
Control can be given to the application, by implementing as much as possible of a protocol stack at the application-level in libraries.
Application-level networking has several advantages:
  - Application-specific performance optimisations can be implemented.
  - Only the protocols that are really needed have to be implemented. On embedded systems specifically, this can save valuable memory.
  - Non-standard transport protocols can be used to communicate between applications with special needs. In the TCP/IP suite UDP [6] is used for unreliable connection-less transport and TCP [7] for reliable connection-oriented transport [1]. There are no alternative transport protocols to choose from. Some applications might benefit from implementing their own transport protocols.
  - The application programmer can decide which API abstraction should be used to encapsulate the protocols. The socket API can be implemented in libraries, if required, or an alternative communication end-point abstraction can be used. That is, the socket API becomes one of many possible APIs.

B. *Existing Systems*

This section introduces related work in the context of the fours categories described above.

(a) Different protocol stack architectures: Clark and Tennenhouse [8] argued that the layered model of network protocols should not force a layered implementation and that "flexible decomposition" should be a "key architectural principle". They suggested *Integrated Layer Processing (ILP)* for efficiency reasons. That is, combining the operations done in different layers into one.

Other systems relax layering constraints by allowing protocol implementations to be composed from "protocol entities" [9] [10]. The x-Kernel [11] works similarly, allowing an object-oriented-like approach, where the relationship between protocol objects with uniform interfaces are defined to create protocol paths through the protocol objects.

However, none of the above systems provide the application programmer with more flexibility.

(b) Different APIs: Fbufs [12] were an efficient way of passing data between levels. This included the passing between system layers and application layers. Banga *et al.* [13] proposed different operating system abstractions to support high performance networking aimed at servers. The x-Kernel system [11] defined an abstraction aimed at improving the performance of "most common patterns of interaction". As described above these systems maintain the encapsulations of the API in system software, thus restricting the application.

(c) Application "steering" of protocols: UNIX STREAMS [14] allows the dynamic introduction of modules into a set of "linearly connected processing modules". Modules can be pushed and popped of a stack of modules. Neighbouring modules communicate through messages; that is, each module provides a routine that accepts messages as an entry point. In this way protocol stacks can be modified by pushing and popping modules on and off the stack. However, any module popped on to the stack has to be pre-linked with the kernel, and is therefore part of the kernel.

The BSD packet filter [15] is an example of UNIX-based systems that allow *packet filters* to be installed in the kernel to capture incoming network packets. These captured packets are then passed directly up to the application, which can perform whatever processing is required. In theory the higher levels of the protocol stack can be implemented at application-level using packet filters.

Plexus [16], a protocol architecture for the SPIN operating system [17] allows the application to "download" protocols as type and pointer-safe extensions into the kernel.

All of the systems mentioned here fail to couple protocol implementation and application tightly. Although application control over network protocol implementation is improved, the implementation is still physically located in system software.

(d) Application-level networking: Thekkath *et al.* [18] designed a system, where protocols were implemented in user-level libraries. The system is based on a microkernel architecture, but protocols are linked with applications. However, the system requires a *registry server* for some operations, such as establishing connections. Therefore, the need for context

switches to trusted system software is not fully eliminated and not all parts of the protocols are tightly-coupled with the application. Nemesis [19] is relatively similar in respect to protocol implementations and has a *flow manager*, which represents the trusted system software part of the protocol stack.

The need for a trusted server process was eliminated with Xok [5], an Exokernel system [20]. The system is based on a library operating system and protocols are fully linked with the application. Only de-multiplexing of incoming packets is done in the kernel.

C. Single-Application Embedded System Model

All the application-level networking solutions described above are aimed at multi-application paradigm systems, where packet de-multiplexing and security issues influence the design of protocol stacks. Embedded systems however, often use a different paradigm. Here, a single-application model is regularly used and although security is important, it is focused at different aspects of the system. Although the system has to be secure from attacks from the outside, different parts of the system are not protected from each other.

Therefore, in this research, a multi-threaded single-application model is used. This relaxes the need for packet de-multiplexing, so that it too can be moved up to the application-level. The application can determine if de-multiplexing of incoming packets between different threads is necessary. Furthermore, buffer management is simplified, with only one queue needed for passing buffers up to user-level. The Arena TCP/IP implementation builds on application-level networking.

III. The Arena Operating System

Arena is an application-oriented operating system [3] [21] intended for both distributed and real-time applications [22]. Operating system policy resides in resource managers implemented as user-level libraries which are linked to the application. The effect of this is to move operating system policy up into the application run-time system. Low-level mechanisms are provided by a hardware-specific nanokernel, or hardware object (HWO). The HWO presents a generic view of low-level processor features. In order to access the low-level mechanisms, resource managers make downcalls to the HWO interface. Conversely, on the occurrence of a hardware event, the HWO can make an upcall to some user-level resource manager. The upcall mechanism enables deferred processing of the event via an application-specific event handler thread. Fig. 1 shows how a hardware interrupt may cause the HWO to invoke the user-level process manager, which schedules a user-level event handler thread. The network policy layer discussed in the present work is implemented as a user-level resource manager. The network policies are accessed via upcalls.

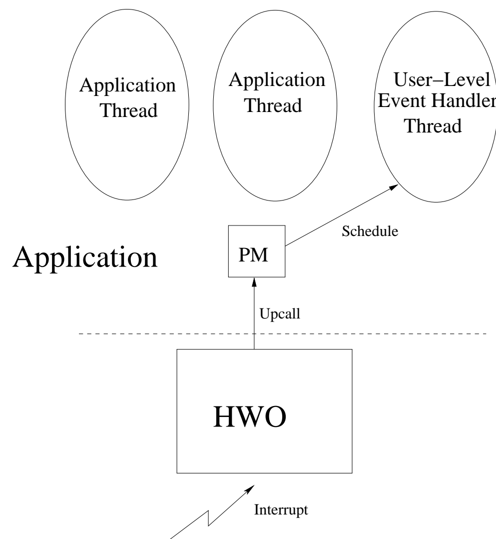

Fig. 1. Arena Event Handling

IV. Mechanism and Policy in Network Protocols

A. Overview

Any protocol implementation on Arena should maintain the Arena philosophy of separating mechanism from policy. Since there are generally more than two layers in network protocol stacks, a decision has to be made which layers should reside in the HWO and which should reside in application libraries. As mentioned in section I, each level uses the mechanisms provided in the level below it to implement a certain amount of policy. It is therefore important to identify layers that introduce *application-specific policy*; that is, policy that should be defined at the end-points of the communication.

This level is termed *application-dependent* and is supported by an *application-independent* level. The application-independent level should be implemented in the HWO, whereas the application-dependent level should be implemented in user-level libraries.

At its simplest, application-independent communication network mechanisms are accessed via, say `send_packet()` and `receive_packet()` primitives. There is no *application-specific policy* involved in these primitives, that is, for example, no decisions are made about the reliability of network communications. A simple application-dependent communications policy relates to, say, having a connection-oriented communication link. At the application-independent level there is no concept of an "established connection". A connection-oriented policy can be implemented on top of a simple send/receive communications mechanism. Decisions about reliability and whether communication is connection-oriented can be implemented at the application-dependent level.

In hard real-time systems, buffering policy is important in order to achieve predictable performance. This means that this buffering of messages presents a problem for the clean separa-

tion of the network mechanisms from application policy. Incoming packets are read from the network by the HWO nanokernel device driver. Although, at this level no application-specific policy should be present, space for the incoming packets has to be allocated. This *buffer management* can be done in different ways. Buffers can be obtained from a fixed pool of statically pre-allocated buffers or can be allocated dynamically. Buffers can be of different size and number. Buffering clearly imposes application-specific policy decisions at the lowest mechanism layer. This impact on application-requirements can be reduced by expanding the donwcall interface to allow the application to “steer” the buffer management policy.

B. TCP/IP

This section looks at TCP/IP, in order to establish which parts of the Arena TCP/IP implementation should reside in the HWO and which parts in application-level libraries. The TCP/IP protocol suite provides protocols above the link layer, and the system described in this paper uses ethernet as the underlying physical and link layer protocol.

Ethernet provides simple ways of sending and receiving packets and a basic naming scheme. No reliability issues are addressed and communication is connection-less. In this respect ethernet is clearly an application-independent protocol.

In the TCP/IP suite itself, IP provides only the mechanisms to send and to receive packets. Connection-related policy decisions are made at the transport layer. That is, by choosing TCP or UDP, reliable connection-oriented policy or unreliable connection-less policy is selected.

Therefore, the Arena TCP/IP implementation has IP in the HWO and TCP and UDP at the application-level. Incoming packets are processed in the HWO up to and including the IP-level, and are then passed to the application through a simple buffer queue. Only one queue is required, because of the single-application model. De-multiplexing of packets to different threads can be done at the application-level. This allows different application-specific policies for de-multiplexing to be used by the application programmer.

ICMP [23] can be regarded as an *error reporting* mechanism, rather than an *error correcting* policy [1]. ARP [24] is regarded as a mechanism for address resolution. ICMP and ARP are therefore implemented in the HWO.

Fig. 2 shows the TCP/IP implementation on Arena and Fig. 3 highlights the *receive case*. It can be seen that all packet de-multiplexing is done at the application level and that a single packet queue is used to pass packets from the HWO to the application. UDP, TCP and packet de-multiplexing run in a user-level event handler thread. The system also leaves the application programmer with the possibility to implement other transport protocols besides TCP and UDP.

V. Buffer Management

As mentioned above, application-specific buffer management policy is trapped at the application-independent level.

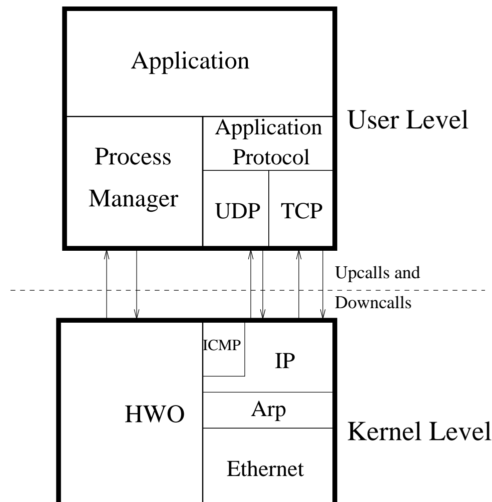

Fig. 2. TCP/IP Implementation on Arena

Therefore, great care has to be taken to allow the application maximum flexibility with regards to buffer allocation. One general principle is that copying of buffers should be avoided for performance reasons. The single-address space nature of many embedded systems facilitates this, because no protection boundaries have to be crossed.

Arena TCP/IP uses the buffer management scheme of lwIP [25], with some slight modifications. This scheme allows two ways of allocating buffers. Buffers can be allocated dynamically through a `malloc()`-like function or from a pool of pre-allocated buffers. This allows the application programmer to use either method of allocating buffers for *sending*.

Receiving however starts at the lowest level, within the con-

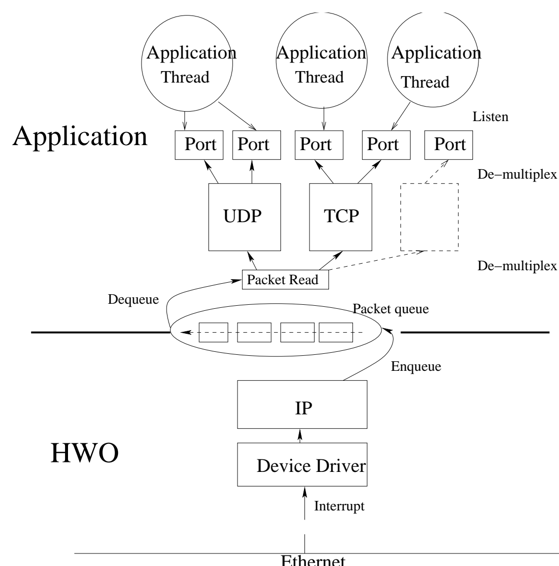

Fig. 3. Receiving of Packets

text of an interrupt handler. The application could influence any buffer allocation policy for incoming packets by using a HWO-provided interface during initialisation. Because usage of pre-allocated buffers is more predictable in terms of execution time, the system described here allocates incoming packets from a pool of pre-allocated buffers. This improves the real-time behaviour of the system. However, the size and number of pre-allocated buffers have to be chosen carefully, in order to avoid wasting scarce memory resources or losing packets because all buffers are in use.

The buffer-management scheme also provides facilities to reserve space for network headers of different level protocols to avoid copying of buffers.

VI. Performance

A. Experimental Setup

All experiments were run on an Atmel AT91M40800-based development board, with 4MB of external RAM and a Cirrus Logic CS8900A 10Mbps ethernet chip. The ethernet chip was used to connect the board to a local ethernet network. Experiments were run at peak and off-peak times, with similar results, suggesting that the processing of packets on the board, rather than network congestion, was the main influence on the timing results.

There were two sets of experiments. The first experiment determined the penalty associated with accessing user-level policy via an upcall, which involves dispatching an Arena event handler thread. The implementation of ICMP was moved from the HWO to user-level. The board was *ping*ed with ICMP handling implemented at user-level or in the HWO, and the round-trip times were measured.

In the second experiment, the time and space efficiency of a transport protocol implemented in a user-level library and ICMP in the HWO was investigated. To simulate a typical use, such as file transfer, a sequence of 2000 UDP packets was sent to the board from a nearby host. Packet size was 256 bytes. The received packets were "bounced back" to the host and round trip time and packet-loss were calculated. The number of pre-allocated buffers was varied to measure its effect on packet-loss. Two scenarios were simulated. In the "non-busy scenario" a single UDP link was used. To simulate a "busy scenario" the board was constantly *ping*ed. The purpose of this experiment was to to optimise the number of pre-allocated buffers.

B. User-Level Policy Overhead

With ICMP implemented in the HWO the *ping*-time was 0.935ms. This increased to 1.051ms, when ICMP processing was moved up into a user-level event handler. This suggest, that locating transport-level policy at user-level increases the packet round trip time by approximately 13.6%. This is mainly due to the thread context switch arising as the result of the upcall. Any system however, has to perform at least one context switch for packets that are passed up to the user-level. Previous work compared networking performance of Arena to that of a microkernel, and found a performance advantage in Arena [26]. These results indicate that the flexibility of the Arena approach does not have an excessive cost.

C. Application-Level Transport Protocols

In the non-busy scenario the round trip time for UDP packets was an average 21ms. Table I shows the results of the packet-loss measurements. It can be seen that the packet-loss remained relatively low and constant, even when only one pre-allocated buffer was available. This suggests that the processing of other incoming buffers, such as broadcasted ARP requests, was fast enough to free the buffers quickly. The probability that a packet is received when the buffers are busy seems very low in the non-busy scenario.

In the "busy scenario" the round-trip time remained an average 21ms, but the packet-loss behaved differently. The results in Table II show that when the number of pre-allocated buffers was reduced, the UDP packet loss remained relatively constant and the *ping* loss rate increased slightly. However, when just one buffer was available, the *ping* packet-loss increased dramatically to 68.75 %. This behaviour can be explained by the fact that the UDP packets are passed up to the application, whereas *ping* ICMP packets are processed only in the interrupt context of the HWO. Furthermore the UDP packets arrived much more frequently than the *ping* ICMP packets. Therefore, UDP packets "blocked" buffer space for much longer and more frequently, increasing the probability that *ping* packets would find no buffer available. The round-trip time for UDP packets remained 21ms on average.

The results suggest that in light use, the amount of memory reserved for pre-allocated buffers can be very low. As little as one buffer for a typical "receive and acknoledge link" can result in acceptable performance. If every pre-allocated buffer is 1520 bytes in size (maximum ethernet packet size plus padding for 4-byte alignment), 10 pre-allocated buffers result in a memory requirement of approximately 15KB.

VII. Future Work

The application-compliant TCP/IP implementation described in this paper is part of a wider research project. The project is concerned with adapting the Arena operating system to support dynamic configuration through run-time code loading techniques. There are two main target areas. Firstly, it is planned to allow the dynamic replacement of OSMs. The second aim focuses on protocol loading. In particular, Grid [27]

TABLE I

Packet loss for a single UDP link

no. buffers	1	2	3	5	10	20
loss (%)	0.13	0.05	0.05	0	0	0

TABLE II

PACKET LOSS FOR A SINGLE UDP LINK PLUS *ping*

no. buffers	1	2	3	5	10	20
UDP (%)	0.17	0.1	0.05	0.05	0.1	0.01
ping (%)	68.75	5.0	2.0	0	0.5	0

protocols could be loaded dynamically onto the embedded system to allow the system to use a computational grid for large computations. The problem with grid computing on embedded systems is the large number of application-level protocols needed. Dynamic loading of only the protocols needed at a given time could facilitate the participation of networked devices in computational grids.

VIII. CONCLUSION

Application-level networking allows flexible and efficient protocol stack implementations that could be particularly useful in networked appliances, because of the embedded and special-purpose nature of such systems.

One such implementation, the TCP/IP protocol stack for the application-compliant Arena operating system, has been described. The system is optimised for embedded systems and is based on a multi-threaded single-application model, resulting in a simpler protocol stack and buffering.

It has been argued that network protocols can be viewed in terms of separating mechanism from policy, in order to achieve application-compliant networking. The Arena TCP/IP protocol stack implements layers that introduce application-specific policy in application libraries, giving the application programmer the flexibility of choosing between policies implemented in different libraries or even implementing application-specific policy.

REFERENCES

[1] D. E. Comer, *Internetworking with TCP/IP*. Prentice-Hall International, 1991.

[2] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, 1984.

[3] K. Mayes, S. Quick, J. Bridgland, and A. Nisbet, “Language- and application-oriented resource management for parallel architectures,” in *ACM SIGOPS European Workshop*, pp. 172–177, 1994.

[4] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implemetation of the 4.3 BSD UNIX Operating System*. Reading, MA: Addison-Wesley, 1989.

[5] G. Ganger, D. Engler, M. F. Kaashoek, H. Briceno, R. Hunt, and T. Pinckney, “Fast and flexible application-level networking on exokernel systems,” *ACM Transactions on Computer Systems*, vol. 20, no. 1, pp. 49–83, 2002.

[6] J. Postel, *User Datagram Protocol- RFC 768*, Aug. 1980.

[7] J. Postel, *Transmission Control Protocol — DARPA Internet Program Protocol Specification – RFC 793*, Sept. 1981.

[8] D. Clark and D. Tennenhouse, “Architectural considerations for a new generation of protocols,” *ACM Computer Communication Review*, vol. 20, no. 4, pp. 200–208, 1990.

[9] C. Tschudin, “Flexible protocol stacks,” in *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pp. 197–205, September 1991.

[10] C. B. Czech, B. Hütter, and M. Gwinner, “Flexible protocol stacks by in-kernel composition,” in *Proccedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1998.

[11] N. C. Hutchinson and L. L. Peterson, “The x-kernel: An architecture for implementing network protocols,” *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, 1991.

[12] P. Druschel and L. L. Peterson, “Fbufs: A high-bandwidth cross-domain transfer facility,” in *Symposium on Operating Systems Principles*, pp. 189–202, 1993.

[13] G. Banga, P. Druschel, and J. C. Mogul, “Better operating system features for faster network servers,” in *Proceedings of the Workshop on Internet Server Performance (held in conjunction with ACM SIGMETRICS ’98)*, (Madison, WI), 1998.

[14] D. M. Ritchie, “A stream input-output system,” *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897–1910, 1984.

[15] S. McCanne and V. Jacobson, “The BSD packet filter: A new architecture for user-level packet capture,” in *USENIX Winter*, pp. 259–270, 1993.

[16] M. E. Fiuczynski and B. N. Bershad, “An extensible protocol architecture for application-specific networking,” in *USENIX Annual Technical Conference*, pp. 55–64, 1996.

[17] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer, “SPIN - an extensible microkernel for application-specific operating system services,” in *ACM SIGOPS European Workshop*, pp. 68–71, 1994.

[18] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska, “Implementing network protocols at user level,” *IEEE/ACM Transactions on Networking*, vol. 1, no. 5, pp. 554–565, 1993.

[19] R. Black, P. Barham, A. Donnelly, and N. Stratford, “Protocol implementation in a vertically structured operating system,” in *IEEE LCN’97*, (Minneapolis, Minnesota), pp. 179–188, IEEE, November 1997.

[20] D. R. Engler, M. F. Kaashoek, and J. O’Toole, “Exokernel: An operating system architecture for application-level resource management,” in *Symposium on Operating Systems Principles*, pp. 251–266, 1995.

[21] R. Morrison, D. Balasubramaniam, M. Greenwood, G. Kirby, K. Mayes, D. Munro, and B. Warboys, “A compliant persistent architecture,” *Software - Practice & Experience, Special Issue on Persistent Object Systems*, vol. 30, no. 4, pp. 363–386, 2000.

[22] S. Kingsbury, K. Mayes, and B. Warboys, “Real-time arena: A user-level operating system for co-operating robots,” in *Proceedings of The Interantional Conference on Prallel and Distributed Processing Techniques and Applications (PDPTA)*, pp. 1844–1850, CSREA Press, 1998.

[23] J. Postel, *Intenet Control Message Protocol — DARPA Internet Program Protocol Specification – RFC 791*, Sept. 1981.

[24] D. C. Plummer, *RFC 826 — An Ethernet Address Resolution Protocol — or — Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*, Nov. 1982.

[25] A. Dunkels, “Minimal TCP/IP implementation with proxy support,” Tech. Rep. T2001-20, Swedish Institute of Computer Science, 2001. http://www.sics.se/∼adam/lwip – access date: 29 July 2002.

[26] K. Mayes, J.Bridgland, S.Quick, and A. Nisbet, “Network performance in arena,” in *Proceedings of HPCN Europe*, pp. 1007–1008, 1996.

[27] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.