KECho - Event Communication for Distributed Kernel Services [*]

Christian Poellabauer, Karsten Schwan, Greg Eisenhauer, and Jiantao Kong

College of Computing,
Georgia Institute of Technology,
Atlanta, GA 30332
{chris, schwan, eisen, jiantao}@cc.gatech.edu

Abstract. Event services have received increased attention as scalable tools for the composition of large-scale, distributed systems, as evidenced by their successful deployment in interactive multimedia applications and scientific collaborative tools. This paper introduces KECho, a kernel-based event service aimed at supporting the coordination among multiple kernel services in distributed systems, typically to provide applications using these services with certain levels of Quality of Service (QoS). The publish/subscribe communication supported by KECho permits components of remote kernels as well as applications to coordinate their operation. The target group of such a kernel-based event service is the rapidly increasing number of extensions that are being added to existing operating systems and are intended to support the Quality of Service and real-time requirements of distributed and embedded applications.

1 Introduction

Kernel-level services and their run-time coordination. The need to offer high or predictable levels of performance, especially in distributed and embedded systems, has resulted in the kernel-level implementation of certain applications and services. Examples include the in-kernel web servers khttpd and tux on Linux, kernel-level QoS management and resource management mechanisms [7], and load balancing algorithms [8]. To attain desired gains in predictable performance, distributed kernel-level extensions must coordinate their operation. For example, for load balancing, multiple machines in a web server cluster must not only exchange information about their respective CPU and device loads (e.g., disks), but must also be able to forward requests to each other without undue involvement of clients and forwarding engines [16]. Similarly, to ensure the timely execution of pipelined sensor or display processing applications in embedded systems, hosts must not only share detailed information on their respective CPU schedules and the operation of the communication links they share [17, 18], but they must also coordinate the ways in which they allocate resources to

[*] This work was partially supported by DARPA/ITO under the Ubiquitous Computing, Quorum, and PCES programs.

pipelined tasks. Finally, the run-time coordination among kernel-level services illustrated above is highly dynamic, involving only those kernel services and machines that currently conduct a shared application-level task. In addition, the extent of such cooperation strongly depends on the application-level quality criteria being sought, ranging from simply 'better performance' to strong properties like 'deadline guarantees.'

Run-time kernel coordination with KECho. This paper presents KECho, a kernel-level publish/subscribe mechanism for run-time coordination among distributed kernel services. Using KECho, any number of kernel-level services on multiple hosts can dynamically join and leave a group of information-sharing, cooperating hosts. Using KECho, services can exchange resource information, share resources (e.g., via request forwarding), and coordinate their operation to meet desired QoS guarantees. KECho uses anonymous event-based notification and data exchange, thereby contrasting it to lower-level mechanisms like kernel-to-kernel socket communications, RPC [9], or the RPC-like active messaging developed in previous work [19]. Furthermore, compared to object-based kernel interactions [20] or to the way in which distributed CORBA, DCOM, or Java objects interact at the user level [10–12], KECho's model of communication provides improved flexibility, since its use of anonymous event notification permits services to interact without explicit knowledge of each others identities.

The *KECho* kernel-level publish/subscribe mechanism shares several important attributes with its user-level counterparts. First, KECho events may be used to notify interested subscribers of internal changes of system state or of external changes captured by the system [4]. Second, it may be used to implement kernel-level coordination among distributed services, perhaps even to complement the application-level coordination implemented with user-level event notification architectures [1–4]. Applications constructed with event-based architectures include peer-to-peer applications like distributed virtual environments, collaborative tools, multiplayer games, and certain real-time control systems. Third, KECho's functionality is in part identical to that of known user-level event systems, which means that we describe it using interchangeable terms like *event notification mechanism*, *event service*, and *publish/subscribe mechanism.* Further, KECho's event services faithfully implement the publish/subscribe paradigm, where events are sent by publishers (or *sources*) directly to all subscribers (or *sinks*). Channel members are anonymous, which implies that members are freed from the necessity to *learn* about dynamically joining and leaving members.

KECho is implemented as an extension to the Linux operating system (using kernel-loadable modules) and offers a lightweight high-performance event service that allows Linux kernel-level services (which could themselves be extensions) to coordinate their actions. The intent is to ensure that distributed applications achieve high/predictable performance and good system utilization. By using the resulting distributed kernel services, distributed applications can improve their use of shared underlying machine resources like processing power and disk space, without having to explicitly interact at the application level. Application-level counterparts to such functionality typically require additional kernel calls and inter-machine communications, and they may even require the implementation of extensions to existing user/kernel interfaces, so that applications can gather the resource information they need from their respective operating system kernels. In contrast, the kernel-level solutions to distributed resource management enabled by KECho can access any kernel or network service and any kernel data structures without restrictions, which is particularly important for fine-grained resource monitoring or control. Finally, KECho can also be used directly by applications, thereby permitting them to directly interact with their distributed components.

Contributions. (1) KECho is an in-kernel event-based group communication mechanism that supports the anonymous and asynchronous cooperation of distributed kernel-based services and user-level applications. (2) KECho (i) achieves high event responsiveness by using a kernel extension that monitors socket activity and (ii) reduces processing and networking overhead by filtering events based on information supplied by the event publisher and by all event sinks. (3) The advantages of the KECho kernel-based communication tool are explained by means of two extensions to the Linux kernel's functionality: (i) a novel resource management system and (ii) a load balancing mechanism for cluster-based web services.

2 Kernel Event Channels

Event notification systems have been used in applications including virtual environments, scientific computing, and real-time control. Compared to user-level implementations of event services, the advantages of a kernel-level implementation include:

- *Performance:* each call to a user-level function of the event system (e.g., residing in statically or dynamically linked libraries associated with the application) can internally result in a high number of system calls. These calls can block, thereby delaying an application and causing unpredictable application behavior. By using a kernel-based service, we can significantly reduce both the number of system calls used in its implementation and the effects on predictability of its execution. Furthermore, if the application components using event services are implemented entirely within the kernel, then no system calls are required at all, and performance is improved further by minimized blocking delays within the kernel. Specifically, a kernel-thread waiting for an event can be invoked immediately after the event occurs, while a user-level application may suffer further delays by waiting in the CPU scheduler's run queue for a time period dependent on its scheduling priority and the current system load.
- *Functionality:* an increasing number of services is being implemented inside of an operating system's kernel, mainly for performance reasons. Only a direct, kernel-to-kernel connection of such services without the additional overheads of user/kernel crossings allows for fine-grained and direct communication and coordination among remote kernel services.

– *Accessibility of resources:* typical user/kernel interfaces restrict the number and type of kernel resources that can be accessed. Kernel-based implementations have no restrictions regarding the access to such resources, that is, resources and kernel data structures (e.g., task structures, file structures) can be accessed and used directly. This allows kernel solutions to make 'smarter' decisions compared to user-level solutions.

2.1 Architecture of KECho

The goal of a kernel-based event service is to support the coordination and communication among distributed operating system services.

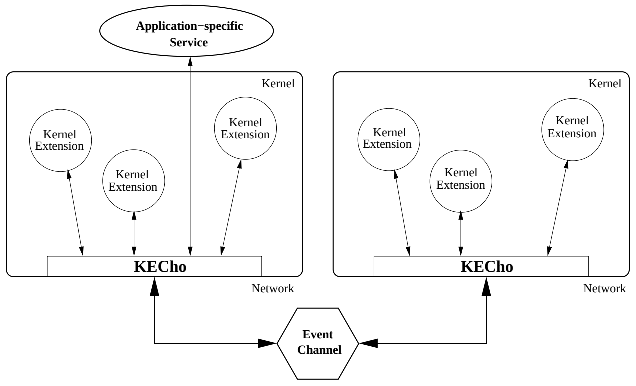


Fig. 1. Event channels in KECho.

Figure 1 shows the architecture of KECho, using which both kernel- and user-level OS services and user-level applications can dynamically create and open event channels, subscribe to these channels as publishers and subscribers, and then submit and receive events. Although the event channel in Figure 1 is depicted as a logically centralized element, it is a distributed entity in practice, where channel members are connected via direct communication links. The channel creator has a prominent role in these communications only in that it serves as the contact point for anyone wishing to join or leave a group. Any number of kernel services can subscribe to an event channel, and events can be typed, the latter meaning that only events that fit a certain description will be forwarded to subscribers.

The implementation of KECho is based on its user-level counterpart, called *ECho* [2], the libraries of which have been ported to six kernel-loadable modules for Linux 2.4.0, each with a certain task:

– *KECho Module:* the main interface to kernel services for channel management and event submission/handling.

- *ECalls Module:* a richer interface to user-level applications that implements a lightweight version of system calls and shared memory segments. In addition, this module can influence CPU scheduling decisions to maximize event responsiveness [5].
- *Group Manager Module:* a user-level *group server*, running on a publicized host, serves as channel registry, where channel creators store their contact information and channel subscribers can retrieve this information. This module supports the communication among subscribers and the group server.
- *Communication Manager Module (CM):* this module is responsible for the connection management, including creating and operating the connections between remote and local channel members. It currently supports TCP connections as well as a reliable version of UDP to which we are planning to add real-time communication properties.
- *Attribute List Module:* this module implements attributes, which are name-value pairs with which performance or QoS information may be piggybacked onto events.
- *Network Monitoring Module (NW-MON):* this module monitors socket activity and notifies the CM module of newly arrived data at any of the sockets associated with an event channel.

2.2 Event Delivery

The lowest module in the KECho module stack, the network monitoring module (NW-MON), allows KECho to register *interest* in certain sockets. Specifically, KECho registers interest in all sockets associated to event channels. NW-MON then will be notified by the network interrupt handler once data arrives at one of these sockets. In return, this module then notifies the CM module of this event.

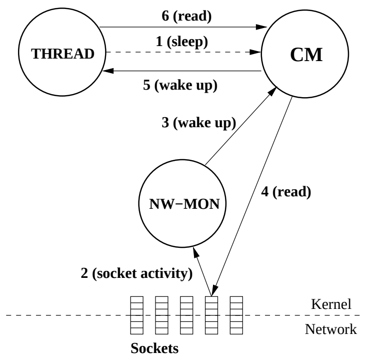


Fig. 2. Event Delivery in KECho.

As an example, a subscriber waits for a new event (step 1 in Figure 2) by sleeping or blocking. Activity of a socket related to an event channel (step 2) prompts the network monitoring module to send a *wake-up* call to the CM module (step 3). CM then reads the data from the socket (step 4) and identifies and notifies (step 5) the thread owning this socket. Finally, the thread can now copy the received data from the CM module (step 6) and act upon this event.

While CM awakens and notifies waiting threads about the arrival of events, it can also *accelerate* event responsiveness by increasing the CPU scheduling priority of the process receiving an event. This is part of the ECalls module and is described in more detail in two other papers [5, 6].

2.3 Filtering

Most event systems offer the possibility to limit the number of events received through *event filters*. Filters can be placed at either the event sink or the event source and can significantly reduce event processing and network overheads. The most basic filters ensure that events are delivered only if they are of certain *types*. Typical event systems allow those filters to base their decisions only on a per-connection basis, where a filter makes its decision without considering the overall channel condition. In addition to event filters, KECho offers so-called *channel filters*, which (1) can be dynamically inserted by the event source and (2) can decide on a per-channel basis which sinks will receive an event, that is, filtering decisions are based on information collected from the publishers and subscribers via separate event channels or via attributes piggybacked onto events. As an example consider the task of load balancing. Here, a service request from a machine in a web server cluster is forwarded to an event channel if the local server is not able to service this request. A filtering function can collect load information from all other servers and then decide which other server will receive the event carrying the forwarded request. Alternatively, if load information is outdated and requests are idempotent, then the quality of load balancing can be improved by simultaneously forwarding the request to n servers, where n is chosen by the event source. Upon delivery of the event to the n best servers (e.g, the servers with the lightest loads) and completed event handling, duplicate responses can be discarded by the load balancing mechanism. In this example, the event source supplies the number of desired recipients of a forwarded request and all event sinks supply their current load information.

A filter can also be applied to incoming events, in which case it is simply invoked once each time an event arrives at the channel. For example, such a filter can decide – based on information from the event source and from all sinks – to which sinks the event will be dispatched. In the load balancing example mentioned above, this kind of filter could make sure that the response to a request is being returned to only the one sink that issued the original request, or it could block multiple responses to the same request. A kernel service can register two filter functions with an event channel, an IN-filter and an OUT-filter (Figure 3). An IN-filter is invoked each time an event is being received by KECho. The IN-filter is able to investigate the event before it is being dispatched to the event sinks. On the other hand, an OUT-filter is being invoked each time

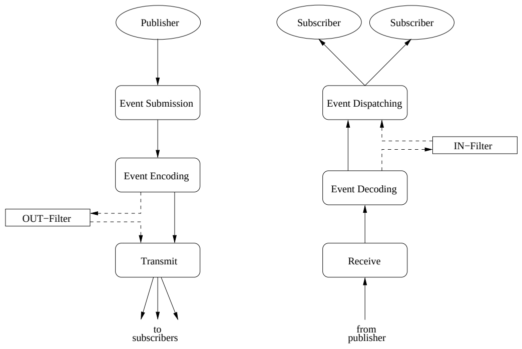


Fig. 3. Channel filters in KECho.

an event is being submitted by a local event source. Again, the filter inspects the event and can decide which remote sinks will ultimately receive the event.

3 Example 1: Resource Management

Applications rely on the availability of certain *system resources* in order to perform their tasks successfully. System resources can include processing power, network bandwidth, disk bandwidth, RAM, and input/output devices such as cameras or printers. Resource management systems [13, 14] have the task to allow applications to discover, allocate, and monitor such distributed resources. This task is made difficult by (i) the dynamic behavior of resources (i.e., resources can join and leave at any time), (ii) the dynamic arrival and departure of application components requiring resources (e.g., through process migration), and (iii) run-time variations in the current resources required by an application.

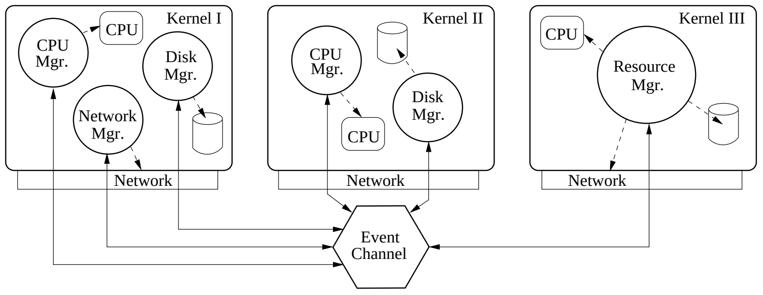


Fig. 4. Resource management with KECho

Figure 4 shows how KECho connects resource managers to facilitate the task of locating and acquiring resources for applications. Kernels I and II have 3 resp.

2 resources that are shared with other hosts, e.g., CPU, disk, and network resources. As an alternative, a kernel could have only one resource manager, which assumes the task of managing all available resources at a host, as shown in kernel III. In both cases, resource managers can forward requests for resource allocations from applications to other, remote resource managers by submitting an event. If a remote resource manager can fulfill the request, it responds accordingly to the manager that forwarded the original request. If there are several positive responses, a resource manager can use certain criteria (e.g., response times, location of the resource) to decide which response to accept or discard. Resource managers can dynamically join or leave resource-sharing groups, by joining a group it makes its resources publicly available to all other members in the group. However, all managers are unaware of the number or the location of other group members and resource requests are submitted and accepted/denied via events.

4 Example 2: Load Balancing

Load balancers in web server clusters [8, 15] have the task to forward requests that can not be handled locally to other servers in the cluster. Figure 5 shows the architecture of a simple load balancing mechanism for a web server cluster.

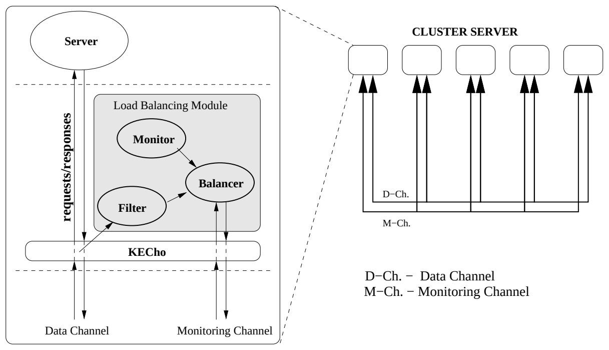


Fig. 5. Load balancing with KECho

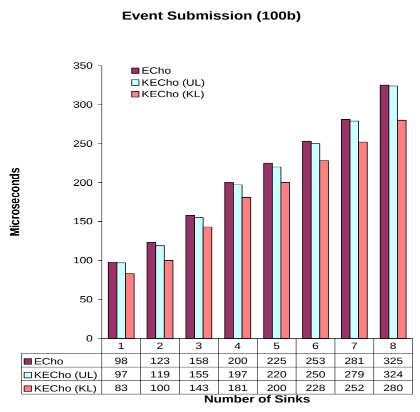
When using KECho to implement load balancing, each server in the cluster subscribes to the shared *data channel*, which is used to forward requests to other servers if the load on the host is too high to successfully handle a request. Further, servers send responses to such requests in form of events over the same event channel. All servers also register two filters, which are supplied by the load balancing mechanism in the kernel: (i) an OUT-filter, which intercepts service requests and decides which remote server(s) will receive a forwarded request, and (ii) an IN-filter, which discards multiple responses from different servers if the request has been forwarded to more than one server. The load balancing decision is based on *load information* exchanged between all servers via a separate event channel, called *monitoring channel*. In addition, the server forwarding a request determines how many remote servers will receive this request. This can improve the server utilization even more if the load information is not updated frequently enough, that is, the n servers with the lowest utilization receive a request and only the first response from these servers will be used, all other responses are discarded.

5 Simulation Results

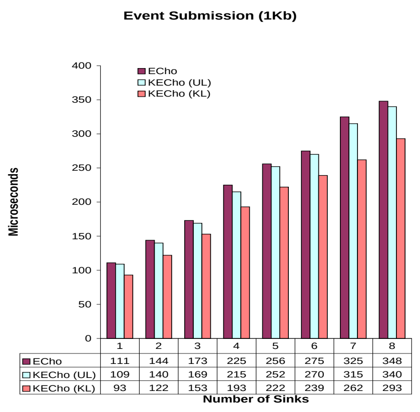
The following microbenchmarks have been performed on a dual-Pentium III with 2x800MHz, 1GB RAM, running Linux 2.4.0. The intent is to investigate the overheads associated with event submission and delivery, channel management, and filtering.

5.1 Event submission

The first measurement compares the event submission overheads of the user-level implementation of event channels (ECho), the kernel-level event channels used by a user-level application (KECho-UL), and the kernel-level event channels used by a kernel-thread (KECho-KL).



	1	2	3	4	5	6	7	8
ECho	98	123	158	200	225	253	281	325
KECho (UL)	97	119	155	197	220	250	279	324
KECho (KL)	83	100	143	181	200	228	252	280



	1	2	3	4	5	6	7	8
ECho	111	144	173	225	256	275	325	348
KECho (UL)	109	140	169	215	252	270	315	340
KECho (KL)	93	122	153	193	222	239	262	293

Fig. 6. Event submission overheads for data sizes of 100 bytes and 1 Kbyte.

The graphs in Figure 6 compare the event submission overheads of these three scenarios for 100b and 1Kbyte, where the overheads of ECho and KECho-UL differ only minimally. This can be explained by the fact that ECho uses only two system calls per sink for the submission of an event, where KECho requires also two system calls, but that number is independent from the number of sinks. Event submissions with KECho-KL show up to 15% (for 100b) and up to 20% (for 1Kb) less overhead compared to ECho.

Table 1. Overheads and number of system calls

	ECho	KECho-UL	KECho-KL
Channel Creation	850μs (56)	182μs (5)	170μs (-)
Channel Opening	approx. 1.5s (117)	approx. 1.5s (5)	approx. 1.5s (-)
Event Submission	100μs (2 per sink)	95μs (2)	85μs (-)
Event Polling	32μs (4)	40μs (2)	5μs (-)

Table 1 compares the performance of some of the functionality of KECho (KECho-UL/KECho-KL) with the performance of the user-level implementation ECho. Channel creation requires 850μs in ECho, compared to 182μs in KECho-UL and 170μs in KECho-KL. The large difference between kernel-level and user-level approach can be explained by the number of system calls required for the creation of a channel in ECho, which is 56, compared to 5 in KECho-UL. The opening of a channel depends on the current number of subscribers, the network transmission delays and other factors, however, typical values for this operation are approximately 1.5s in all three cases. Event submission takes about 100μs per event subscriber for ECho, compared to 95μs and 85μs for KECho-UL and KECho-UL, respectively. In ECho, the overhead for polling for new events is 32μs (4 system calls) compared to 40μs (2 system calls) in KECho-UL. The reason for this increase are some inefficiencies in the implementation which will be addressed in our future work. However, the overhead for event polling in KECho-KL decreases to only 5μs. Note that while typical applications using ECho have to periodically poll for new events, KECho is able to notify kernel threads almost immediately of the arrival of a new event. This ability is investigated in the following section.

5.2 Event Delivery

Events in KECho are pushed from event sources to event sinks. The network monitoring module of KECho is able to immediately notify a waiting thread of the arrival of such an event. Typical latencies measured from the arrival of an event at a socket to the invocation of a handler function are in the range of 250-300μs. In the case of ECho and KECho-UL, these latencies depend heavily on the polling frequency, the systems load, and the scheduling priority of the application receiving the event. However, ECalls ability to *boost* the scheduling priority of an application that receives a newly arrived event can significantly reduce these latencies. This cooperation between ECalls and the CPU scheduler is described in detail in [5].

5.3 Filtering Overhead

The following measurements have been performed on a cluster of 4x200MHz Pentium Pros, with 512MB RAM, connected via 100Mbps Ethernet, running Linux 2.4.0.

The filtering functions (IN- and OUT-filter) serve to reduce processing and network overhead depending on application-specific attributes, supplied by the event producer and the event subscribers.

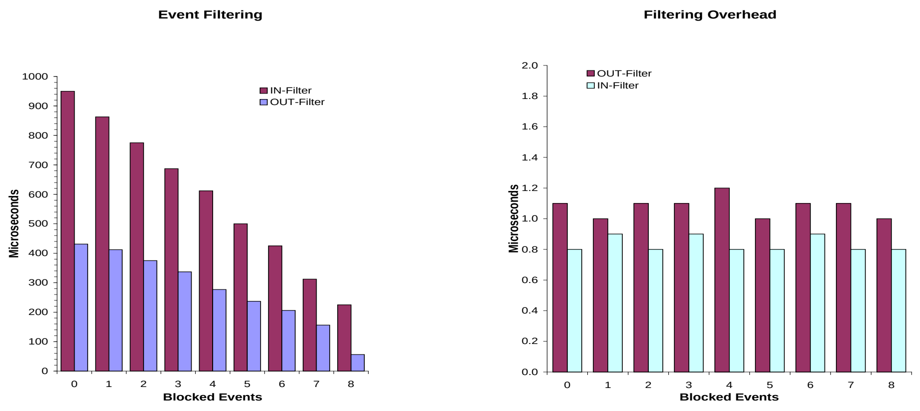


Fig. 7. Filtering of events can reduce event submission and event handling overheads (a), while the filtering overhead is only in the microsecond range (b).

The left graph in Figure 7 compares the advantages of event filtering with IN- and OUT-filters. The left bars show the event handling overhead for a host with 8 sinks, i.e., an incoming event is dispatched to all 8 sinks and the overhead is approximately 950μs (event handling in this example means copying of the incoming event into a buffer and printing a time-stamp into a file). This overhead can be reduced significantly when we use an IN-filter to block the event from being dispatched to all 8 sinks, e.g., if only one sink receives the event, the overhead is reduced to 312μs. If the filter blocks the event completely (i.e., the event is discarded), the overhead is a little more than 200μs. The right bars in the same graph compare a similar scenario, however, the overhead shown in the graph is the overhead associated with event submission, when the number of remote sinks is 8. The overhead in this example is 430μs. However, when an OUT-filter is being used to block the submission of the event to some servers, this overhead can be reduced, e.g., if the event is submitted to only one sink, the overhead is 156μs. If the event is discarded (i.e., no sink will receive the event), the overhead is 56μs. The second graph in Figure 7 compares the overhead of the IN- and OUT-filters that have been used for the results in the left graph. Both the IN-filter and the OUT-filter use a number of simple if-else statements to decide if an event has to be submitted/dispatched to a certain sink or not. The overheads are independent of the number of events submitted or blocked and are very low in the example shown here, e.g., approximately 1μs for the OUT-filter and 0.9μs for the IN-filter.

5.4 Simulated Web Server Results

In this section we investigate the load balancing mechanism introduced in Section 4 in more detail. Measurements have been performed on a cluster of 8 nodes, acting as a web server cluster. Web servers receive requests at rates ranging from 20 to 50 requests per second. Each request requires a simulated web server to perform processing for approximately 38ms. The first graph in Figure 8 shows the response times (in milliseconds) without any load balancing compared to the scenario where load balancing is being used. Requests in this experiment have a time-out of 5s, leading to the leveling off at 5s of the first line in the graph, i.e., requests are either being handled within 5s after request receipt or discarded otherwise. In the second scenario, we modify the server such that requests that have been waiting for more than 2.5s are being forwarded to other servers in the cluster. In these experiments, we assume that there is at least one server in the cluster with utilization less than 10%.

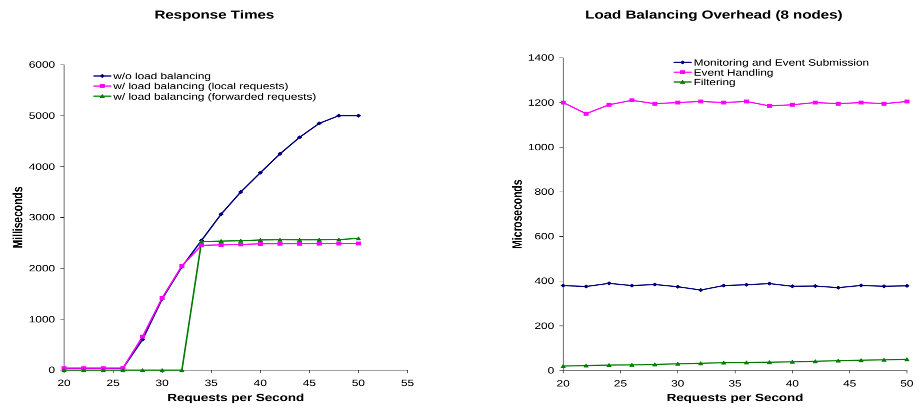


Fig. 8. (a) Response times for a simulated web server cluster and (b) overheads of the load balancing mechanism used in this experiment.

The second line in the graph (w/ load balancing - local requests) shows the response times of all requests which are handled on the local node. This time, the response times level off at 2.5 at request rates of approximately 33 per second. The third line shows the response time of the requests being handled on remote servers, which is slightly higher than the times measured at the local server due to the overhead of two events being submitted and received (forwarded request and request response). The second graph in Figure 8 analyzes the overhead for the load balancing mechanism, which makes sure that only one other server (dependent on load information collected from these servers) will receive the forwarded request. The graph compares the overhead of three actions performed by the load balancing mechanism: (i) the monitoring of CPU utilization and the submission of events carrying this information, (ii) the handling of incoming CPU information from other servers in the cluster, and (iii) the filtering necessary to ensure the delivery of the forwarded request to the server with the lowest utilization. The graph shows that all these overheads vary only minimally with the number of requests, where the task of event handling is the most expensive (approximately 70% of the total load balancing overhead).

The final experiment investigates the advantage of event filtering in more detail. The OUT-filter introduced above forwards requests to the servers with low load to ensure small response times. However, the frequency of load information exchange among the nodes in a server cluster has an obvious influence on the load balancing quality, i.e., if load information is not exchanged frequently enough, the forwarding decision can be based on outdated information, which reduces the effectiveness of load balancing.

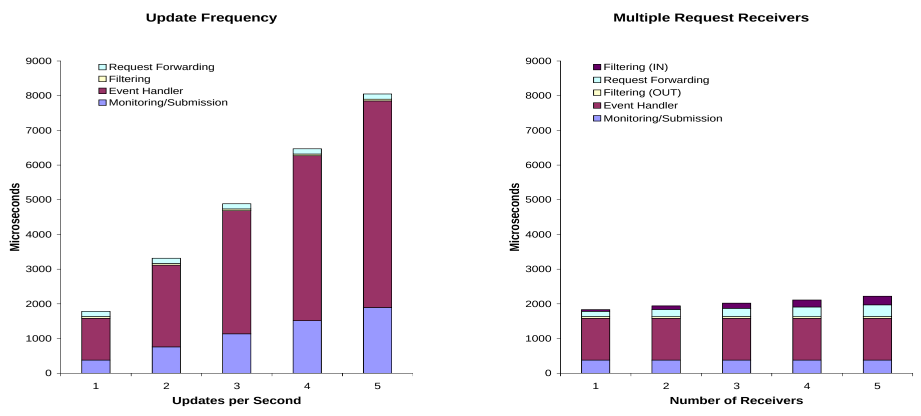


Fig. 9. Comparison of update frequency of load information (a) and forwarding of events to more than 1 server in the cluster (b).

The left graph in Figure 9 compares the overhead of load balancing dependent on the frequency of load information events. The overhead is mainly due to the event handling process, followed by the load monitoring and event submission process. Smaller overheads are caused by the actual forwarding of the requests and the filtering functions. The overhead increases rapidly with the number of events exchanged per second, e.g., more than 8ms with a frequency of 5 events per second. The right graph in Figure 9 compares the approach, where the frequency of load events is kept constantly at 1 per second, however, the filter forwards the request to up to 5 different servers. In other words, multiple servers in the cluster respond to the event and only the first response is being used by the server that issued the event carrying the forwarded request. Again, the event handling and the load monitoring and event submission contribute most to the overheads, however, the overhead increases only minimally with the number of event sinks. The biggest increase in overhead is caused by the IN-filter, which has the task of discarding duplicate responses. This experiment ignores the increased total utilization in the whole cluster due to the request handling by multiple servers. As an alternative to the solution suggested above, a server could issue a *cancel event* to all other servers, that makes sure that only one server handles a request. If several servers issue a cancel event, a time-stamp or some other criterion can decide which server wins. This approach reduces the unnecessary processing on the servers, however it increases the event communication by up to n cancel events per forwarded request.

6 Conclusions

The need for globally managing system services is exemplified by previous work on distributed resource management, on load balancing, and QoS mechanisms. This paper addresses dynamic service management by providing a novel facility for inter-service cooperation in distributed systems. KECho is a kernel-based publish/subscribe communication tool that supports anonymous and asynchronous group communication. KECho's main components are its lightweight interface to user-level service realizations, a network monitor that minimizes the latency of event delivery, and channel filters that allow kernel services and applications to intercept event submissions with the goal of minimizing network traffic and optimizing system performance.

Our future work will investigate the two examples introduced in this paper in more depth and analyze their performance compared to user-level solutions. Further, we will extend KECho to support real-time events, thereby addressing the substantial set of applications requiring real-time guarantees. In addition, we will deploy KECho in the embedded, wireless system domains for which its ability to access and use power, load, and network information is critical to the success of this class of ubiquitous applications. Finally, while protection issues have been ignored to this point, we are already investigating and implementing protection mechanisms that will ensure the proper system operation in face of misbehaving kernel extensions.

References

1. A. Rowstron, A-M. Kermarrec, P. Druschel, M. Castro: SCRIBE: The Design of a Large-scale Event Notification Structure. Proc. of the 3rd Intl. Workshop on Networked Group Communications, London, UK, 2001.
2. G. Eisenhauer, F. Bustamente, K. Schwan: Event Services for High Performance Computing. Proc. of High Performance Distributed Computing, 2000.
3. T. H. Harrison, D. L. Levine, D. C. Schmidt: The Design and Performance of a Real-time CORBA Object Event Service. Proc. of the OOPSLA '97 Conference, Atlanta, Georgia, October 1997 .
4. C. Ma, J. Bacon: COBEA: A CORBA-Based Event Architecture. Proc. of the Fourth USENIX Conf. on Object-Oriented Technologies, Santa Fe, New Mexico, April 1998.
5. C. Poellabauer, K. Schwan, R. West: Coordinated CPU and Event Scheduling for Distributed Multimedia Applications. Proc. of the 9th ACM Multimedia Conference, Ottawa, Canada, October 2001.
6. C. Poellabauer, K. Schwan, R. West: Lightweight Kernel/User Communication for Real-Time and Multimedia Applications. Proc. of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001), Port Jefferson, NY, June 2001.
7. T. Plagemann, V. Goebel, P. Halvorsen, O. Anshus: Operating System Support for Multimedia Systems. The Computer Communications Journal, Elsevier, Vol. 23, No. 3, February 2000, pp. 267-289.
8. A. Bestavros, M. Crovella, J. Liu, D. Martin: Distributed Packet Rewriting and its Application to Scalable Web Server Architectures. Proc. of the 6th IEEE International Conference on Network Protocols, Austin, TX, October 1998.
9. A. D. Birrell, B. J. Nelson: Implementing Remote Procedure Calls. ACM Transactions on Computer Systems, 2(1), February 1984.
10. Object Management Group: CORBAservices: Common Object Services Specification, July 1997, (http://www.omg.org/).
11. D. Box: Understanding COM. Addison-Wesley, Reading, MA, 1997.
12. A. Wollrath, R. Riggs, J. Waldo: A Distributed Object Model for the Java System. USENIX Computing Systems, vol. 9, November/December 1996.
13. F. Kon, T. Yamane, C. K. Hess, R. H. Campbell, M. D. Mickunas: Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. Proc. of USENIX COOTS 2001.
14. I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy: A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In International Workshop on Quality of Service, 1999.
15. V. Cardellini, M. Colajanni, P. S. Yu: Dynamic Load Balancing on Web-server Systems. IEEE Internet Computing, Vol. 3, No. 3, May/June 1999.
16. M. Aron, D. Sanders, P. Druschel, W. Zwaenepoel: Scalable Content-aware Request Distribution in Cluster-based Network Servers. Proc. of the USENIX 2000 Annual Technical Conference, San Diego, CA, June 2000.
17. D. Ivan-Rosu, K. Schwan: FARA– A Framework for Adaptive Resource Allocation in Complex Real-Time Systems. Proc. of the IEEE Real-Time Technology and Applications Symposium, June 1998.
18. D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, J. Walpole: A Feedback-Driven Proportion Allocator for Real-Rate Scheduling. Proc. of the Third Symposium on Operating System Design and Implementation, New Orleans, February 1999.

19. T. von Eicken, D. Culler, S. Goldstein, K. Schauser: Active Messages: A Mechanism for Integrated Communication and Computation. Proc. of the 19th International Symposium on Computer Architecture, pages 256–266, May 1992.
20. G. Hamilton and P. Kougiouris: The Spring Nucleus: A Microkernel for Objects. Report Number: TR-93-14, April 1993.