Self-Adapting Concurrency: The DMonA Architecture*

Sam Michiels, Lieven Desmet,
Nico Janssens, Tom Mahieu, Pierre Verbaeten
DistriNet, Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
Sam.Michiels@cs.kuleuven.ac.be

ABSTRACT

A major problem in today's Internet servers is that they suffer from extreme peak loads. Traditional (operating) systems are designed to perform extremely well under heavy load conditions. However, it is not feasible to over-provision resources only to support peak loads. A key factor to deal with such peak loads is internal concurrency control. We have developed a component based architecture (DMonA), which allows to adapt internal concurrency according to measured throughput. Performance tests show that DMonA outperforms traditional approaches, while it is still very manageable thanks to the underlying DiPS component architecture.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming—*Parallel programming*; D.2.11 [Software Architectures]: [Domain-specific architectures]; D.4.8 [Operating Systems]: Performance—*Monitors, Simulation*

General Terms

Design, Performance

Keywords

Component architecture, self-adaptation, concurrency

1. INTRODUCTION

Today's Internet servers (such as web or FTP servers) can be perfectly optimized to serve huge numbers of clients simultaneously. The Google search engine for example serves 150 million requests, daily [6]. However, such systems are seldom optimized to deal with peak loads. The well-known

*Research for this paper has been carried out in order of Alcatel Bell with financial support of the Flemish institute for the advancement of scientific-technological research in the industry (IWT SCAN # 010319).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WOSS '02, Nov 18-19, 2002 Charleston, SC, USA.
Copyright 2002 ACM 1-58113-609-9/02/0011 ...\$5.00.

“Slash-dot Effect” shows that it is not uncommon to experience more than 100-fold increases in demand when a site becomes popular [20]. It is not feasible to install a server farm that is big enough to handle the maximal potential demand. However, systems should be able to detect overload situations and to handle peak loads gracefully.

We propose the DiPS (DistriNet Protocol Stack) [9] component architecture as a solution to develop *manageable* system software such as file systems or protocol stacks. DiPS separates functionality from other aspects such as inter-component communication and internal parallelism. We argue that such separation of concerns allows to manage specific aspects (such as concurrency) more easily. Our goal is to build a system from reusable components that can adapt itself to cope with changing circumstances.

In this paper we present DMonA, the DiPS Monitor Architecture. DMonA is a monitor and management extension to DiPS, which allows to detect performance bottlenecks, propose solutions to the problem and deploy these solutions at run-time. We apply DMonA to adjust internal concurrency dynamically, depending on measured throughput in specific *areas* in the system.

2. BACKGROUND AND RELATED WORK

Different research areas collide in this paper: self-adaptability, software engineering practices for concurrent programming, separation of concerns and software architectures. The combination of these research domains results in a self-adapting concurrency software architecture.

Self-Adaptability. Lots of research projects tackle software adaptability in all its flavors. A possible taxonomy classifies adaptability based on two orthogonal criteria: the initiator of adaptation and the time of adaptation. The former category deals with adaptability initiated by a human administrator, an application or the system itself. The latter category deals with adaptation at design, build, installation, boot or run-time.

This paper focuses on run-time adaptation initiated by the system itself (*self-adaptation*). Moreover, software support is provided for determining which part of the system needs to be adapted and how. This requires *self-monitoring*. Seltzer [16], [17] presents similar ideas on self-monitoring and self-adaptation.

Concurrency and separation of concerns. A critical element in our concurrency research is the separation of concurrency from functional code [12], [10]. Kiczales [8] defines non-functional concerns as *aspects* that cross-cut functional code. An aspect (for example concurrency) is written in

Figure 1: A DiPS Component

a specific aspect language and is woven into the functional code by a so-called aspect weaver at pre-processing time. Although this approach clearly separates all aspects from the functional code (at design-time), all aspects disappear at run-time, what makes it very difficult (if not impossible) to adapt aspects dynamically. Apertos [5] introduces *concurrent objects* that separate mechanisms of synchronization, scheduling and interrupt mask handling from the functional code (such as a device driver). Therefore, programmers can concentrate on writing functional code without having to write auxiliary code for synchronization [4]. This makes software more understandable, which reduces the risk of errors.

Self-adapting concurrency. Research on adaptability often focuses on adapting the system's *functionality*. This is also one of our research tracks [7]. However, the contribution of this paper is in the run-time adaptability of the concurrency model. Therefore, we present separated *concurrency units* that are adaptable at run-time. Similar work is done in SEDA [23, 22]. SEDA divides the system in independent *stages* that consist of an incoming event queue, a thread pool and an application-supplied event handler. The stage's operation is managed by the controller that adjusts resource allocations and scheduling dynamically. The main difference with our approach is that we provide management and resource control as an orthogonal extension, which can be removed when appropriate.

Software architectures. The DiPS architecture is a heterogeneous software architecture. It combines architectural styles such as the layered, pipe and filter, blackboard and event-based style [3]. DiPS offers a *layered* architecture such as Scout [14]. However, it also allows to zoom in into component and connector level. DiPS offers three component communication mechanisms. First, the basic mechanism is forwarding packets between components via connectors. From such a *pipe and filter* architectural view point, DiPS can best be compared with the Click modular router [15]. Second, DiPS packets can be annotated with so-called meta-information which can be inspected by other components. This results in a *blackboard* communication architecture for anonymous inter-component communication. Finally, DiPS supports *event based* communication through specific event connectors. A similar but fully event driven architecture for protocol stacks is Cactus [24].

An additional DiPS advantage is that concurrency can be introduced into a pipeline by using specific concurrency connectors. Such connectors decouple concurrency from functionality in the components, as will be explained later.

3. DIPS COMPONENT ARCHITECTURE

DiPS [9, 10] is a domain-specific heterogeneous software architecture for developing system software such as file sys-

Figure 2: A DiPS pipe for a simple file system

tems or protocol stacks. The main goal of DiPS is to support reuse and adaptation at design-time [11, 18, 19] as well as at run-time [7]. It therefore clearly separates different aspects such as interaction and concurrency from each other [12].

DiPS basics. A DiPS component can be seen as a building block (**Unit**) surrounded with an explicit entry and exit point (resp. **PacketReceiver** and **PacketForwarder**). Figure 1 shows a basic DiPS component. The **Unit** encapsulates the *characteristics* of a component.

Data is sent through DiPS components as first class **Packets**. A **Packet** not only represents pure data, but also *meta-information* such as request size, request target, session information or request type. This meta-information allows anonymous blackboard component communication. Moreover, it is crucial to be able to monitor and interpret requests.

A DiPS system is created by connecting DiPS components into a pipe (see figure 2). Different connection mechanisms are supported. A pipe connection just connects two components. An active concurrency unit (ACU) introduces concurrency into a DiPS pipe. It encapsulates a request queue and a scheduler. This way, the request flow is broken into parallel pipes. A dispatcher unit (the black dot in figure 2) dispatches requests into different branches of the pipe based on meta-information in the request.

Separation of concerns in DiPS. It is important to decouple the **PacketReceiver** and the **PacketForwarder** from a **Unit**. First, they separate a **Unit** from the way its enclosing component is connected with other components. By connecting the **PacketForwarder** from one component with the **PacketReceiver** from another, two components are coupled while their internal **Units** are independent. Second, by having an explicit **PacketReceiver** and **PacketForwarder**, incoming data can be intercepted transparently, before and after a **Unit**. This is essential to monitor the data flow or to block incoming requests while a unit is being replaced at run-time [7].

To stress the conceptual difference [8] between functionality and concurrency, we differentiate between different unit types (**FunctionalUnit** and **ConcurrencyUnit**). Concurrency code is separated into **ConcurrencyUnits**, which allows programmers to focus on pure functionality without being bothered about concurrency.

4. DIPS MONITOR ARCHITECTURE

DMonA is a monitoring and management extension of the DiPS component architecture. It takes advantage of existing DiPS abstractions (such as explicit component entry and exit points or the **Packet** abstraction). Management can be divided in three sub tasks: detecting problems (*self-monitoring*), proposing solutions to the problem (*adaptability strategies*) and deploying this solution in the DiPS functional layer (*self-healing*).

4.1 Goals

The primary goals of DMonA are as follows.

- **DiPS + DMonA = self-adapting system software.** The combination of DiPS and DMonA must result in a self-adapting system. DMonA must be generic enough to be applied in different contexts. As a consequence, DMonA must be designed as a modular system where monitoring sensors and policies can be changed according to the context.
- **DMonA as orthogonal DiPS extension.** The DMonA management layer must be designed as an orthogonal (independent) DiPS extension. This means that DMonA can be added or removed without *disturbing* DiPS components at the functional level. The advantage is that all legacy units can be reused for interaction monitoring (such as throughput monitoring and access pattern recognition). However, if DMonA needs internal unit state, which is not offered publicly, (legacy) units must be extended with a *management interface*. Remark however that this extension is very localized and independent from other units.
- **Reuse.** Not only legacy units must be reusable, also monitoring sensors and decision strategies must be reusable in different contexts. For example, throughput sensors must be reusable independent from the functional level.

4.2 Self-monitoring: sensors

DMonA monitors the underlying functional DiPS layer. For example, to monitor throughput, incoming and outgoing requests of a unit¹ are counted. To monitor I/O access patterns, specific meta-information about I/O characteristics (such as request type (read/write), request size or request target) is collected.

To gather information from the functional layer into the management layer **sensors** are used. The **Monitor** and its coupling with the functional DiPS layer is shown in figure 3. We differentiate between two types of sensors: sensors that collect information about the internal state of a **Unit** (**State Sensors**, SS) and sensors that collect information about the messages that flow through the system (**Analysis Sensor**, AS). State sensors are invoked via the sensor interface (SI) of a **Unit**. Analysis sensors are subscribed as listeners at the **PacketReceiver** and **PacketForwarder**.

The **Monitor** plays a central role in DMonA. All the information that is gathered by the sensors is processed at regular times by the **Monitor**. To decouple the decision policy (*when* is done *what*) from the core monitor (where all the information is gathered), we introduce a separate monitor strategy. Different monitor strategies can be plugged into the monitor (as we will show in section 5).

4.3 Adaptability strategies

Based on the information gathered from the functional level, DMonA decides what actions should be taken to improve the system's performance. For example, if the arrival rate of a unit is much higher than the departure rate increasing parallelism would be a good adaptability strategy.

¹Monitoring can take more than 1 unit into account. To focus discussion we will not discuss this explicitly.

Figure 3: Management model

If I/O requests are mainly small read requests which tend to be sequential in nature, activating caching and pre-fetching would increase performance. However, when read requests tend to be random, pre-fetching is better turned off.

Remark that a **Monitor** only has local responsibility. To monitor the system globally, individual **Monitors** have to be coordinated by a *meta-monitor*. This prevents that local monitor strategies become counter productive. For example, a local concurrency adaptability strategy cannot distribute threads appropriately without a global view on the system. The meta-monitor will actively gather information from the individual **Monitors** and processes this information to announce a global decision. This results in specific tasks which are delegated to individual monitors.

Currently, the active behavior is mainly shifted towards the meta-monitor to avoid ping-pong effects between the meta-monitor and the local monitors. However, it can be interesting to allow local monitors to have limited active behavior when this is synchronized with the meta-level. This is not yet provided.

4.4 Self-healing

After a problem has been detected by the monitor and a solution has been proposed by an adaptability strategy, the system is ready to be adapted. Manageable units at the functional layer provide a *management interface* (MI) that is used by DMonA to activate the changes that are proposed by the adaptability strategy (see figure 3).

To allow communication from the management layer to the functional (DiPS) layer, we provide manageable **Units** with two interfaces: a sensor interface to get internal state information from a unit and a management interface to set parameters and strategies. There is no explicit communication from the functional layer into the management layer. Sensors decouple both layers.

5. DMONA CONCURRENCY PROTOTYPE

We have applied DMonA for dynamically optimizing thread distribution in a variably loaded DiPS system. Since concurrency is separated from functionality in DiPS, we can purely focus on DiPS active concurrency units.

5.1 Active Concurrency unit

Figure 4: DiPS ACU model

Figure 4 shows the internals of a DiPS active concurrency unit (ACU) with a `PacketQueue` and one or more `PacketSchedulers`.

PacketQueue. Incoming requests are stored in the `PacketQueue`. The capacity of the queue can be increased to cope with request bursts or it can be decreased to release memory in favor of other queues. The design of the `PacketQueue` also provides customizable strategies to handle buffer underflow or overflow (`OverflowStrategy` and `UnderflowStrategy`). Possible strategies are adapting the size of the queue, re-sorting requests or removing low priority requests.

PacketScheduler. The `PacketScheduler` is a thread that fetches a request from the `PacketQueue` and guides it through the component pipe until the next ACU. The `PacketScheduler` communicates with the ACU via the scheduler interface (SCHI on figure 4). Because starting up a thread is an expensive operation, `PacketSchedulers` are made *reusable* and can be migrated from one ACU to another at run-time.

5.2 Throughput monitoring

Figure 5 shows the DMonA management level for a DiPS ACU. To enable throughput monitoring three sensors are used. Two analysis sensors will count the number of incoming (AS1) and outgoing (AS2) requests. They can be reset by the monitor to initiate a monitoring phase. The state sensor (SS3) collects information about the capacity of the queue and the amount of requests that is stored. The `ACMonitor` collects this sensor data at regular times via the `ACSensorInterface` (ACSI on figure 5).

5.3 Concurrency Adaptability Strategy

Based on the information in the sensors, the `ACMonitorStrategy` decides *when and how* the behavior of the ACU should be adapted to improve performance. For example, `PacketSchedulers` can be added or removed, `PacketQueue` size can be increased or decreased or overflow and underflow strategies can be changed.

In this prototype local ACU monitors execute tasks from the meta-monitor, which is the only active management part. We present a meta-monitor strategy with two tasks: performance bottleneck detection and `PacketScheduler` distribution over different ACUs. More complex strategies could also selectively drop requests based on available meta-information in a request.

Bottleneck detection. A bottleneck can be described as an area (one or more components) where the request arrival rate is higher than the outgoing rate (i.e. more arriving requests than can be processed). There can be more than one bottleneck, but the algorithm searches for the bottleneck with the smallest throughput.

Figure 5: Management layer for a DiPS ACU

Similarly, the algorithm searches for areas with an arrival rate that is smaller than the departure rate (i.e. areas that can process more requests than they actually get). Those areas are candidates to free one or more schedulers in favor of the overloaded area.

Scheduler distribution. The meta monitor strategy optimizes throughput by distributing all available schedulers (N) in such a way that processing power is balanced between *underloaded* and *overloaded* component pipes.

The `SchedulerManager` (see figure 5) is responsible for adding or removing a `PacketScheduler` from an ACU. The `SchedulerManager` will be triggered by the monitor when it decides that a re-allocation of `PacketSchedulers` is appropriate.

Changing the number of associated `PacketSchedulers` is done via the `SchedulerManager`. Other DMonA actions are performed directly through the `ACManagementInterface` of the ACU (ACMI on figure 5).

The details of the thread distribution algorithm can be found in [2]. This algorithm searches for an optimal allocation of schedulers over the ACUs. However, such an algorithm is likely to result in a stable but sub-optimal distribution. This is the case when scheduler distribution is optimal following the thread distribution algorithm, although bottlenecks can be found. To prevent this, DMonA disturbs an *optimal* distribution from time to time by adding an extra scheduler in an area and re-initiating the distribution algorithm. This technique of using excitations to migrate from one stable state to another, more optimal, stable state is well-known in biology and chemistry [21].

6. PERFORMANCE RESULTS

This section shows the benefits of using DMonA compared to other concurrency models. Three concurrency models are compared. First, the commonly used *thread-per-request* model, which assigns a new thread to every incoming request. Second, the *bounded thread pool* model, which fetches a thread for every incoming request from a thread pool with fixed size. When the number of parallel requests exceeds a limit, additional requests are rejected. This model is for instance used in web servers such as Apache [1] and IIS [13]. We have implemented this model by using an unmonitored DiPS ACU with a fixed number of `PacketSchedulers` (threads). The third model is the *monitored ACU* model, which has been presented in this paper.

The test setup consists of a component pipe with three components. Each component represents an *area* in the sys-

	Area A	Area B	Area C
Phase 1 (0 - 399)	100	700	100
Phase 2 (400 - 799)	200	200	600
Phase 3 (800 - 1199)	100	700	100

Table 1: Dynamically changing area load in 3 phases of each 400 seconds

Figure 6: Comparison of three concurrency methods handling increasing processing load. Thread-per-request is performing better until 100 concurrent requests. For increasing load, throughput quickly drops below the monitored and unmonitored ACU bounded thread pool models.

tem with a certain load that varies dynamically. Each area is managed by an ACU, which is again managed by a meta-monitor. The monitor-strategy is the same as described in section 5.3. The test identifies three phases in which the load of each area is changed (shown in table 1). Each phase takes 400 seconds. In the first phase, area B is much more loaded than the other areas. In phase two, the bottleneck is shifted towards area C. Phase three again puts all the load in area B.

All software (implementing DiPS, DMonA and performance tests) has been written in Java 2 (build 1.4.0-b92) with green threads. Although Java performance cannot be compared directly with native system throughput, the tests do allow to compare different approaches in specific situations.

6.1 Peak loads

Figure 6 shows system throughput when processing load linearly increases over time. The graph shows that the thread-per-request model is performing best until 100 parallel requests. For more requests, thread switching overhead swallows all processing time and throughput quickly drops below the monitored and unmonitored ACU model.

The difference between the unmonitored and the monitored ACU model is due to a better `PacketScheduler` distribution in the monitored ACU model. The DMonA benefit grows as load increases.

6.2 Variable load

To show the benefits of using DMonA, we apply it in a context where internal load varies dynamically. A real-life example of such a situation could be a file system where

Figure 7: Measured throughput in 3 areas with changing load. Thread switching overhead soon becomes the bottleneck for the 'thread per request' model. The unmonitored ACU model suffers from unbalanced thread distribution. The monitored ACU model outperforms the two other models although throughput varies due to a non-stable scheduler distribution.

I/O is characterized by alternating bursts of cached and non cached requests. The latter results in the cache-miss component pipe (see also figure 2) being alternately loaded and unloaded.

DMonA throughput outperforms the other models over the three phases (see figure 7). The unmonitored ACU model performance changes depending on the load in the three areas. The peak performance at 800 seconds (see figure 7) can be explained as follows. During phase two, requests are queued in the ACU before area C because this area is heavily loaded. During phase three, when processing load in area C suddenly drops, all buffered requests for this area are released quickly which results in a peak throughput. However, once those queued requests have been processed, throughput drops again. Throughput of the thread-per-request model is again very low because the amount of parallel requests is very high.

Figure 7 also shows the ping-pong effect of monitored ACU throughput (throughput deviation from the average of 10 requests per second). This is because the margins used to decide to add or remove a `PacketScheduler` are too narrow. This results in continuously moving `PacketSchedulers` from one ACU to another. We are aware that the current monitor strategy is not optimal. Our main goal was to show DMonA benefits in practice. This goal has been realized.

7. CONCLUSIONS

This paper presents the DMonA management and monitoring architecture on top of the DiPS component architecture. Thanks to DiPS abstractions such as separated concurrency units and separated component entry and exit points, DMonA could be developed as an *orthogonal DiPS extension*.

Although this paper focuses on adapting internal parallelism, we have also applied DMonA to build a self-adapting file system prototype. DMonA was used to recognize I/O access patterns and to adapt caching strategies accordingly. We were able to *reuse* the basic `Monitor` and interaction

sensors. The DMonA design allowed to introduce specific I/O request sensors and monitor strategies.

Performance tests have shown that DMonA gracefully handles peak requests and variable internal load. Although the current concurrency adaptability strategy is not perfect, performance tests show a clear benefit compared to static approaches. Moreover, application of DMonA in two prototypes was straightforward which convinced us that DiPS and DMonA is a strong combination for developing *self-adapting system software*.

8. REFERENCES

- [1] Apache Software Foundation. The apache web server. <http://www.apache.org>.
- [2] L. Desmet. Adaptive System Software with the DiPS Component Framework. M. Sc. Thesis. K.U.Leuven, Dept. Comp. Science, May 2002. In Dutch.
- [3] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, 1993.
- [4] J. ichiro Itoh and Y. Yokote. Concurrent Object-Oriented Device Driver Programming in Apertos Operating System. Sony Computer Science Laboratory Technical Memo SCSL-TM-94-005, Department of Computer Science, Keio University, 1994.
- [5] J. ichiro Itoh, Y. Yokote, and M. Tokoro. SCONE: Using Concurrent Objects for Low-level Operating System Programming. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 385–398, 1995.
- [6] G. Inc. Google's technical highlights. <http://www.google.com/press/highlights.html>, 2001.
- [7] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework. In *Proceedings - The Seventh International Workshop on Component Oriented Programming*, 2002.
- [8] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.
- [9] F. Matthijs. *Component Framework Technology for Protocol Stacks*. PhD thesis, K.U.Leuven, Department of Computer Science, Dec. 1999.
- [10] S. Michiels, P. Kenens, F. Matthijs, D. Walravens, Y. Berbers, and P. Verbaeten. Component Framework Support for developing Device Drivers. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, volume 1, pages 117–126. FESB, 2000.
- [11] S. Michiels, T. Mahieu, F. Matthijs, and P. Verbaeten. Dynamic Protocol Stack Composition: Protocol Independent Addressing. In *Proceedings of ECOOP-OOOSWS'2001 - 4th ECOOP Workshop on Object-Oriented and Operating Systems*, 2001.
- [12] S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten. DiPS: A Unifying Approach for Developing System Software. In A. D. Williams, editor, *Proceedings - The Eighth Workshop on Hot Topics in Operating Systems*, page 175. University of Karlsruhe, IEEE Computer Society, 2001.
- [13] Microsoft Corporation. IIS 5.0 Overview. <http://www.microsoft.com/windows2000/techinfo/howitworks/iis/iis5techoverview.asp>.
- [14] A. B. Montz, D. Mosberger, S. W. O'Malley, and L. L. Peterson. Scout: A communications-oriented operating system. In IEEE, editor, *Proceedings of Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 58–61, 1995.
- [15] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of Symposium on Operating Systems Principles 1999*, pages 217–231, 1999.
- [16] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Issues in Extensible Operating Systems. Computer Science Technical Report TR-18-97, Harvard University, November 1997.
- [17] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. In IEEE, editor, *The Sixth Workshop on Hot Topics in Operating Systems: May 5–6, 1997, Cape Cod, Massachusetts*, pages 124–129. IEEE Computer Society Press, 1997.
- [18] I. Şora, F. Matthijs, Y. Berbers, and P. Verbaeten. Automatic Composition of Systems from Components with Anonymous Dependencies Specified by Semantic-Unaware Properties. In *Proceedings of Technology of Object-Oriented Languages and Systems Eastern Europe (TOOLSEE) 2001*. IEEE Computer Society Press, March 2002.
- [19] I. Şora, P. Verbaeten, and Y. Berbers. Using Component Composition for Self-customizable Systems. In *Proceedings of Workshop On Component-Based Software Engineering: Composing Systems from Components*, pages 23–26. IEEE, 2002.
- [20] L. A. Wald and S. Schwarz. The 1999 Southern California Seismic Network Bulletin. Seismological Research Letters, 71(4), July/August 2000.
- [21] M. M. Waldrop. *Complexity. The emerging science at the edge of order and chaos*. Touchstone, 1993.
- [22] M. Welsh. The Staged Event-Driven Architecture for Highly Concurrent Server Applications, 2000.
- [23] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, 2001.
- [24] G. T. Wong, M. A. Hiltunen, and R. D. Schlichting. A configurable and extensible transport protocol. In *Proceedings of INFOCOM 2001*, pages 319–328, 2001.