Appears in Proceedings of the 10th ACM SIGOPS European Workshop (EW-2002), pp. 213-218, Saint-Émilion, France, September 2002

A Utility-Centered Approach to Building Dependable Infrastructure Services

George Candea and Armando Fox
Stanford University, Gates 452
Stanford, CA 94305, U.S.A.
{candea,fox}@cs.stanford.edu

Abstract

Achieving dependability in large scale infrastructure systems always requires making intelligent tradeoffs. This paper draws upon ideas from economics and operations research to propose a systematic approach to thinking about such tradeoffs in terms of the system beneficiary's utility. The design process consists of choosing a spanning set of axes for the design space, explicitly formulating utility functions with respect to each axis of the spanning set, and then iteratively converging on the design that maximizes overall utility. We apply this process to the design of a fictitious online banking system.

1. Introduction

In this paper we describe a process for designing infrastructure services, particularly those that are Internet-based. Examples of such services include airline reservations (Sabre), e-commerce (Amazon), wide area caching (Akamai), searching (Google), portals (Yahoo), instant messaging (AOL), news sites (CNN), web e-mail (Hotmail), online auctions (EBay), etc.

We view dependability of a service as an expression of how well the system's properties match the system's requirements. Successful infrastructure services require significant amount of functionality, maximum correctness, must meet usability and maintainability requirements, be performant, secure, highly available, inexpensive to develop, etc. Decades of computer engineering have demonstrated the difficulty of simultaneously achieving all these properties; therefore, making smart engineering tradeoffs is vital to dependability. Paraphrasing [15], we define the dependability of an infrastructure service to be the degree in which a match between *required* and *provided* levels of availability, reliability, safety, and security has been achieved.

The importance of properly reconciling service quality, availability, performance, security, etc. increases commensurately with the system's scale. In small systems, achieving the right "mix" is typically an optimization, whereas in giant scale systems [5], sound tradeoffs become indispensable to the very possibility of building these systems. For example, airline seat reservation systems, faced with the practical impossibility of detecting duplicate bookings in real time, delegate such analysis to offline or off-peak hours, trading consistency for large transaction volume and high availability. RAID-5 systems, when scaled to hundreds of disks, have an increased probability of experiencing multiple simultaneous disk failures, which make the entire storage system fail. This led to the invention of RAID-6, which trades performance for the ability of tolerating double failures, hence decreasing the probability of data loss by 2-3 orders of magnitude [7].

There is extensive literature dealing with pairwise tradeoffs, but in practice tradeoffs are always made along more than two axes at any given time. For example, the Inktomi search engine allows for incomplete results to be returned in response to a search query, in order to obtain in exchange higher performance, higher availability, and decreased system cost. Akamai's content distribution network has cache nodes distributed worldwide to improve its level of performance, availability, and cost, but in exchange the system trades security and manageability. Finally, when Yahoo chose to implement its own hash table-like data store, instead of using an off-the-shelf database, it traded cost and portability for higher performance and more appropriate functionality. All these tradeoffs move the designed system closer to the service's requirements; without these design choices, the services would likely not have survived.

Software engineers are well aware of multiway tradeoffs employed in building computer systems, but they seldomly make these tradeoffs explicit. Consequently, building dependable infrastructure services is still an art. This paper proposes a way to bring this art one step closer to engineering, through the definition of:

- a simple model for the space in which design tradeoffs are made,
- a simple, comprehensive vocabulary for describing properties that result from these tradeoffs, and

- a step-by-step process for trading system properties against each other, such that overall system utility is maximized.

2. The Design Process

System properties can usually be described in terms of a small set of "design axes". Some of these are rather universal, like data consistency, performance, availability, while others may be application-specific, like data lifetime, security, and interactivity. The process of designing a system amounts to attempting to maximize the overall utility of the system with respect to these properties, which we can envision as axes of a design space. We employ the utility function concept, as used in economics, to model the level of "happiness" that the beneficiary of an infrastructure system derives from different levels of the system's properties.

Consider the following process:

1. Identify a coordinate system for the design space, i.e., a set of axes that span the design space. The notion of spanning set is used loosely to mean that any interesting tradeoff can be expressed in terms of the axes in the spanning set. Moreover, the axes need to be orthogonal, i.e., we cannot express one of them as a combination of the other axes. For example, security is orthogonal to performance and availability, whereas availability is not orthogonal to time-to-fail and time-to-repair.

2. Formulate what is typically called a "requirements specification" in terms of these axes, usually the result of discussions between the client and the system vendor. Specifically, articulate utility functions with respect to each of the spanning axes, expressing how useful a given level of that property might be.

3. Identify major design regions within the design space, akin to equivalence classes in design space, in which all designs have a common pattern. For example, "three-tiered Internet service architecture" would denote one such design region. For each design region, choose an exponent consisting of a representative design.

4. For each region exponent, find its coordinates (or ranges of coordinates) in design space. Based on these coordinates and the utility functions, compute the overall utility of that exponent. The implication is that the utility of the exponent is representative of the utilities of all designs in that region.

5. Choose the design region whose exponent has the highest overall utility. If the design is sufficiently specific, proceed to build it. Otherwise, go back to step 3 and choose subregions of the chosen design region and drill down into more detail.

To illustrate this process, in the sections that follow we present a mock design process for an online bank. Our treatment is more qualitative than quantitative; the use of specific numbers does not imply rigorousness. Our main purpose is to provide the intuition for the proposed process, rather than advocate specific axes or utility functions for the chosen application domain.

3. The Coordinate System

For the banking application we choose five axes: quality of data, service availability, performance, security, and total cost of ownership.

Quality of data reflects how "good" that data is to the user application. While generally this axis would incorporate both the notion of consistency and fidelity, for our simplified banking example we look only at consistency between displayed results and the golden copy of the account, stored in the bank's database. The quality axis is continuous in nature and ranges from 0% to 100%.

Availability captures the percentage of read and write requests that are completed satisfactorily by the service over the lifetime of that service, i.e., the probability that a given request will be satisfactorily answered [9]. The availability axis is continuous in nature and ranges from 0% to 100%. Some services measure availability as the percentage of time they are available to reply to requests, regardless of whether such requests are issued or not; we believe such a workload-independent definition is inaccurate.

Security usually encompasses authentication, authorization, confidentiality, integrity, and accountability, which can be treated separately, if needed. In this example, we will take the approach of using an ISO standard, ITSEC [1], which takes all aspects of security into account when evaluating system security. The ITSEC, likely the most successful computer security evaluation system, was developed after the Orange Book and is closer to today's technology. There are 7 ITSEC evaluation assurance levels (EAL). For example, EAL 3 corresponds to a system methodically tested and checked for security vulnerabilities, with grey box testing and selective independent confirmation of developer test results. The toughest level, EAL 7, requires that a system's design be formally verified and tested, with the formal model supplemented by a formal presentation of the functional specification and high level design, showing correspondence between design and implementation. There exist a variety of certified commercial products, such as the Cisco Secure PIX Firewall 5.2 (EAL 4), Oracle 8i (EAL 4), Sun Solaris 8 (EAL 4), Hitachi MULT-OS v3 (EAL 6), etc. The security axis is discrete, taking on EAL values from 0 to 7.

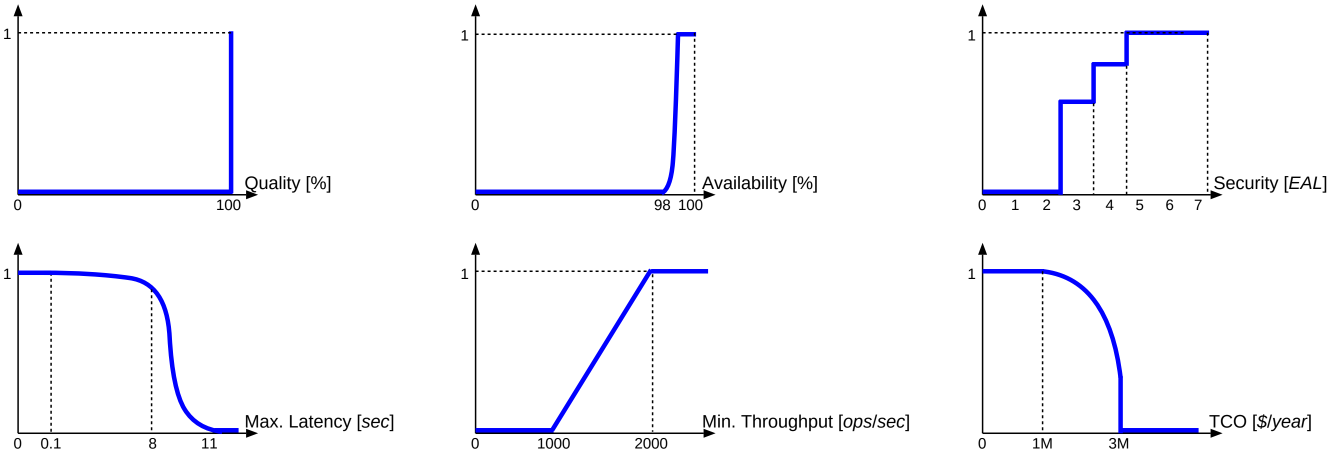

Figure 1. Normalized utility functions for an example online banking application.

Other possible quantifications of security include a simple set containment approach, in which higher levels of security incorporate a larger set of security precautions included in the system. Another approach, depending on the application, is to quantify security according to cryptographic key sizes [17]. [20] proposes a way to evaluate and quantify the security of storage systems.

Performance is usually viewed as an expression of the throughput and latency of access. For a general banking service we may want to consider both read and update throughput/latency and perhaps even differentiate based on the particular data set being accessed. However, in this example we will only look at a general measure of overall throughput and latency. The performance axes have a continuous value set; throughput is measured in operations/second, and latency in seconds.

Total cost of ownership (TCO) includes hardware/software costs, training, maintenance, technical support, network connectivity, etc. In this example we use the per-year amortized cost, with the TCO axis quantified in dollars/year.

When choosing values and metrics for points on any of the design axes, system designers will generally choose units specific to the applications that will use the state repository they are building.

4. The Requirements Specification

The requirements specification is a collection of utility functions, one for each axis of the design space, along with a formula for combining individual utilities into an overall utility. It is acceptable for points on the axes to not be quantified with absolute metrics; what really matters is that values can be compared to each other. The units used for measuring utility need to be uniform across all five axes, to be able to correctly compare utilities throughout the design space.

Utility functions can be specified at various levels of detail, from qualitative graphs to precise quantitative functions. The right level of detail is generally obtained at the end of an iterative process, in which utility functions are successively refined. The general approach we propose for building these utility functions is to choose salient points and then qualitatively interpolate between them.

In this example, the formula for combining utilities is simply a multiplication.

Quality: Certain applications, such as large search engines, routinely reduce completeness of their answers [5], that, however, would be unacceptable in the case of a banking application, where consistency between the reported balances/payments/etc. with the true bank account is crucial. Therefore, the only salient point in this case is the 100% quality point, and Figure 1 shows one of the simplest utility curves possible: a step function. Any quality below 100% is useless, hence utility 0; once the quality is 100%, it perfectly meets the requirement of the application.

Availability of the service is the percentage of requests that are satisfactorily fulfilled by the bank's web site. According to various surveys, the true availability of the best-of-breed web sites today is on the order of 98%, so we choose that as one salient point. For competitive reasons, we would expect the online bank to find a system with poorer availability than 98% to be totally useless. The utility of availability rapidly increases until it reaches the order of 3 nines, after which any further improvements in availability become rather worthless, as we can count on users to retry a failed request. This yields a second salient point at 99.9%. We interpolate and show the resulting curve in Figure 1.

Security of the service is defined in terms of how useful the different assurance levels would be, so it is is natural to choose the assurance levels as salient points. We could say any security less than EAL 3 would be considered inappropriate for a bank, while an EAL of 5 or above is plentiful—beyond a certain point, other factors become of greater concern than the security of the service itself (e.g., disgruntled employees, administrator mistakes, etc.) The levels of 3, 4, and 5 naturally correspond to different levels of utility.

Performance of the service is described by latency and throughput. When determining the salient points for latency, we can resort to research that looks at how web response times affect the user experience. For example, [2] shows that the web site performance threshold at which customers get frustrated and leave is between 8-11 seconds. Another study [4] shows that response times of 100 msec or less give the feeling of instantaneous response, hence any response time between 0-100 msec should have utility 1. We interpolate between these points and obtain the latency utility curve shown in Figure 1. For throughput, in our example, we arbitrarily decide that a value below 1,000 operations/second is worthless and any throughput of 2,000 or more operations/second is sufficient. To interpolate between these two salient points we use the observation in [2] that there is a linear 50 percent relationship between site performance and site abandonment.

Generating a utility function for *total cost of ownership* is a highly non-technical task. Assume, for the sake of illustration, that the approved IT budget for this project is $1M for each year of operation, and the total IT budget for all of the bank's operations is $3M/year. Therefore, as long as TCO stays below $1M, the utility of the resulting system is maximized with respect to cost. It is possible to stretch this cost up to $3M (at decreasing utility), beyond which it is impossible to support the system.

5. Finding A Global Maximum

In this section we walk through the iterative process of converging toward a design that maximizes overall utility. We repeatedly identify regions of the design space and compute their expected range of overall utility. In the first phase, there is a wide variety of representative implementations for the type of services described here; we choose two of them for illustration.

System 1 is a completely distributed design. The US customer base is served by five geographically distributed clusters of web servers. Clients are routed to their nearest front end via DNS mappings. Behind these front ends lie application servers implementing the bank's business logic; each application server converses with a local replica of a distributed database. Any time updates are made, they are geographically distributed to all replicas using some efficient mechanism, such as log-based replication.

System 2 is a totally centralized design. One single data center serves all customers, a single database holds all the acount information, and there is a single cluster of application server instances in the middle tier. A load balancer distributes requests to the web servers in the front end, that communicate with the application server.

In the table below we estimate in each column the utility we would get from each system along each axis of the design coordinate system. For example, the quality for both systems, since they are implemented on top of ACID databases, is always 100%, hence a utility of 1. Security utility for the first type of system is 0, because one cannot presently assemble such a distributed system from software components certified at EAL 3 or better. TCO is better for the second system because of it being centralized and thus easier to manage. The numbers shown are somewhat arbitrary—we chose them to illustrate distinguishing features, rather than prescribe certain utility values for online banks.

Region	Quality	Availability	Security	Latency	Throughput	TCO	Overall
1	1.0	0.9 - 1.0	0	0.9 - 1.0	0.9 - 1.0	0.5 - 0.7	0
2	1.0	0.2 - 0.4	0 - 1.0	0.8 - 0.9	0.6 - 0.8	0.7 - 0.9	0 - 0.26

To compute the overall utility range, we multiply the individual utilities. Based on the results, we choose the design region that contains totally centralized systems and proceed to the second phase. In the chosen region, we refine the *System 2* type into two other representative design points:

System 2.1: For the data center we choose a Sun Solaris 8 platform, running the Oracle 8i database server in conjunction with BEA's WebLogic 7.0 application server. The front ends are Netscape Enterprise 3.6 web servers.

System 2.2: The platform is Redhat Linux 7.2. We hire an outside company to implement a custom database that offers specific performance and functionality properties not found in commercial databases. The middle tier is entirely developed in house, and the web front ends consist of Apache 2.0 web servers.

Similar to the first phase, we construct a table with the axis-specific utilities of the two possible choices.

System	Quality	Availability	Security	Latency	Throughput	TCO	Overall
2.1	1.0	0.2 - 0.4	0.5 - 1.0	0.8	0.8	0.7 - 0.8	0.05 - 0.21
2.2	1.0	0.3 - 0.4	0 - 0.5	0.9	0.8	0.8 - 0.9	0 - 0.13

Based on these estimates, we choose the first option, *System 2.1*. We only showed these first two phases, but the process would not stop here—it would proceed with further refinement of the chosen system, including various configuration aspects, choices for finer grain components, etc.

6. Discussion

The example presented here is very simplistic, as our intention was to illustrate a way of thinking, rather than give specifics of the online banking system. For clarity, we normalized all utility functions and assumed that they are equally important to the final computation. When this is not the case, one can simply scale each utility function appropriately (e.g., if security is more important than other properties, its maximum utility could be 5 instead of 1). Alternatively, we can give different weights to the utilities in the combining formula.

6.1. The Design Hyperspace

The six axes used in the above example form a 6-dimensional space; each possible implementation of the desired system corresponds to a point in this space. For each point in this space, we can compute its overall utility. If we consider utility as a seventh axis for this design space, then the implementations with their utilities describe a discrete "utility manifold" in 7-dimensional space. Making tradeoffs consists of navigating this manifold in search for the global utility maximum, a point at which tradeoffs are optimal given the utility functions. As utility functions change (varying user demands, market pressures, etc.), the utility manifold changes in shape; as technology changes, new points may appear or disappear in the design space.

It is difficult to reason in terms of "navigating" a manifold in 7-dimensional space, which is why designers reason mostly in terms of pairwise tradeoffs. However, any one property often affects two or more of the other properties, so the overall design process does take place in this 7-dimensional space—a fact that must be recognized and incorporated in our software development methods.

6.2. Global Plateaus, Not Maxima

In the process shown above, we made the simplifying assumption that the utility manifold is smooth, i.e., no "cliffs" are encountered when moving from one point to another. Such cliffs do exist, however [10]: for example, when a front end node is hit with high traffic, there is a point at which it starts thrashing, causing performance to drop all of a sudden. The true aim of the design process is therefore not an absolute global maximum, but rather a global plateau, which provides both a high overall utility and a high tolerance to disturbances.

If such cliffs are present within a region, choosing an exponent with a representative utility for the region is more difficult—it may become necessary to choose smaller regions or evaluate multiple exponents for each region. The number of phases in the design process is proportional to the smoothness of the utility manifold: the fewer cliffs, the fewer phases. Moreover, the shape of the utility manifold in the neighborhood of a chosen point can reveal some valuable tradeoffs that were overlooked, thus guiding the designer in choosing neighboring points to explore.

6.3. Dynamic Runtime Tradeoffs

Unpredictable workload is the norm in large scale infrastructures, and over-provisioning to handle all possible load spikes is most of the time too costly [3]; fast dynamic tradeoffs are therefore required. Some, like the public telephone system, trade availability for quality by blocking the initiation of calls in overload situations. Others, such as CNN.com, reduce richness of web pages to keep availability constant during high load periods [16]. We believe our utility-based approach is well-suited to building adaptable systems that make tradeoffs at runtime, e.g., by changing operating parameters. Operators can express requirements through the utility functions, and the system autonomously changes system parameters to maximize utility, without requiring human anticipation or authorization of the tradeoffs. A challenge in applying this method to machine-chosen tradeoffs is the need to have concrete metrics for the axes and explicit utility functions.

6.4. Utility Functions Are Hard to Formulate

The design process is complex and almost always includes multiple interactions with the target system's beneficiary, to understand what the requirements really are. It is difficult to make the requirements explicit, and often clients themselves do not understand their intended use of the system. For example, consultants at Oracle Corp. are often asked by clients to build a data warehouse that is available 24x7. In many cases, however, it turns out that the client's updates and accesses will be run in batch mode, and so the usefulness of 24x7 availability is not much higher than a somewhat more reduced level, yet the cost of taking the data warehouse to that level is significant [19]. Luckily, stating utility functions is easier for systems that both the clients and the providers have had previous experience with.

Even if many clients do not know what their utility functions are, these functions do exist. They need to be made explicit, if the resulting system is to be dependable. As suggested in [13], well-designed graphical tools can guide users in enunciating requirements, as well as provide "what-if" analyses to confirm the functions are valid.

6.5. Related Work

In choosing economic tools and concepts to represent customer requirements, we were inspired by the architecture for Internet service levels proposed by [21] as well as by the market-based approach to resource allocation described in [11]. The field of operations research has already developed an extensive theory [12] on the use of utilities in making decisions and value tradeoffs, which we intend to peruse in our future work. Value analysis [18] is a well-established engineering technique that provides a disciplined, step-by-step approach to identifying and removing unnecessary cost in product and service design—a similar approach to what we are trying to develop.

Hippodrome [3] employed an iterative design process to configure storage systems, similar in spirit to what we described here. [6] used a utility function approach for energy and server resources in large data centers. Extensive work has also been done in identifying and making pairwise tradeoffs in systems; two notable examples include Bayou [8] and TACT [22]. The idea of exploring global maxima in a design landscape forms the basis of genetic programming [14]; in maximizing a fitness function (rather than an overall utility function), genetic algorithms use the principles of Darwinian natural selection to converge onto an optimal solution.

7. Conclusion

In this paper we argued that tradeoffs in computer system design always take place in a multidimensional space, rather than just along two axes. Understanding this fact and having a suitable model is particularly important in large scale infrastructure services, where the right tradeoffs are critical to the very existence of the service. We described an iterative, high level process based on utility functions that can help in better matching system properties to the beneficiary's requirements, hence improving system dependability. We illustrated this process with a simple example of choosing the right commercial software for a banking service; the same process also applies to the development of code.

8. Acknowledgements

We would like to thank the Stanford/Berkeley Recovery Oriented Computing team and Eric Anderson, Dan Candea, Andy Chou, Kevin Fu, James Hamilton, Kim Keeton, Costa Sapuntzakis, Alfred Spector, Marvin Theimer, and Amin Vahdat for valuable insights and comments on the material presented here.

References

[1] Information technology security evaluation criteria. ISO-15408.

[2] Tying performance to profit. Technical Report STP01-C04, Jupiter Media Metrix (Jupiter Research), New York, NY, June 2001.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the Conference on File and Storage Technologies (FAST-02)*, pages 175–188, Monterey, CA, 2002.

[4] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *Ninth International World Wide Web Conference (WWW9)*, Amsterdam, The Netherlands, May 2000.

[5] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

[6] J. Chase, D. Anderson, P. Thakar, and A. Vahdat. Managing energy and server resources in hosting centers. In *Proceedings of the 18thACM Symposium on Operating Systems Principles*, pages 103–116, Banff, Canada, 2001.

[7] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[8] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the 1994 Workshop on Mobile Computing Systems and Applications*, December 1994.

[9] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, San Francisco, CA, 1993.

[10] S. D. Gribble. Robustness in complex systems. In *Eighth Intl. Symposium on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau, Germany, May 2001.

[11] K. Harty and D. Cheriton. A market approach to operating system memory allocation. In *Market-Based Control: A Paradigm for Distributed Resource Allocation*, Singapore, 1996. World Scientific Publishing Co.

[12] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley & Sons, 1976.

[13] K. Keeton and J. Wilkes. Automating data dependability. In *Proceedings of the SIGOPS European Workshop (2002)*, Saint-Emilion, France, Sep 2002.

[14] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[15] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, Vienna, Austria, Dec 1991.

[16] W. LeFebvre. CNN.com – facing a world crisis. Invited talk at USENIX LISA, San Diego, CA, Dec 2001.

[17] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, Sep 2001.

[18] L. D. Miles. *Techniques of Value Analyis and Engineering*. McGraw-Hill, 1972. 2nd edition.

[19] M. Moy. Personal Communication, 2002.

[20] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proceedings of the Conference on File and Storage Technology*, pages 15–30, Monterey, CA, Jan 2002.

[21] S. Shenker. Fundamental design issues for the future internet. *IEEE Journal on Selected Areas in Communications*, 13(7):1176–1188, Sep 1995.

[22] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 305–318, Oct. 2000.