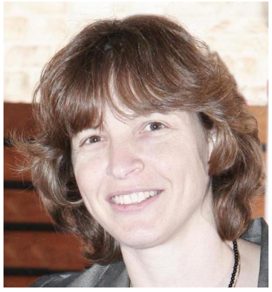
ACM SIGACT News Distributed Computing Column 30

On Distributed Computing Principles in Systems Research

Idit Keidar
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
idish@ee.technion.ac.il



Distributed systems are increasingly deployed in the real world nowadays. Concepts like high-availability, disaster recovery, service-oriented architectures, grid computing, and peer-to-peer are now a standard part of a software engineer's vocabulary. Data is often stored on remote servers or disks, and redundancy is employed for fault-tolerance and high availability. Even computations on a single machine are becoming increasingly parallel due to the advent of multi-core architectures, as discussed in the previous Distributed Computing Column.

Not surprisingly, the "systems" research community follows a similar trend, and increasingly focuses on distributed systems, distributed storage, and parallel computing. Topics like replication and fault-tolerance, (including Byzantine fault-tolerance), which have been studied in distributed computing conferences like PODC and DISC for a couple of decades, have now found their way to the mainstream of systems research. Even the SIGOPS Hall of Fame Award, which recognizes the most influential Operating Systems papers, was awarded in 2007 to five distributed computing papers (see below). At the same time, new research topics with a distributed flavor have emerged in response to real-world drives such as peer-to-peer applications, data storage across multiple administrative trust domains, and multi-core architectures.

This column examines how distributed computing principles can (and do) come into play in systems research. I first list the laureates of the 2007 SIGOPS Hall of Fame Award. Next, Allen Clement surveys recent papers in systems conferences (SOSP and OSDI) that employ distributed algorithms. He first discusses how topics that have been studied in the distributed algorithms community are now used in systems research, and then overviews new topics that are treated in both communities, albeit differently. Allen also points out where future research on foundations of distributed computing can help advance the field.

The bulk of this column is by Roy Friedman, Anne-Marie Kermarrec, and Michel Raynal, who discuss the important principle of modularity in distributed systems. They illustrate how this principle has contributed to research in the areas of shared memory-based computing and agreement problems. They then advocate a similar approach for peer-to-peer systems.

Many thanks to Allen, Roy, Anne-Marie, and Michel for their contributions to this column. Upcoming columns will focus on "quantum computers meet distributed computing" and on "teaching concurrency".

Call for contributions: Please send me suggestions for material to include in this column, including news and authors willing to write a guest column or to review an event related to distributed computing. In particular, I welcome tutorials and surveys, either exposing the community to new and interesting topics, or providing new insight on well-studied topics by organizing them in new ways. I also welcome open problems with a short description of context and motivation.

2007 SIGOPS Hall of Fame Award

The SIGOPS web page[1] stipulates that "The SIGOPS Hall of Fame Award was instituted in 2005 to recognize the most influential Operating Systems papers that have appeared in the peer-reviewed literature at least ten years in the past." In 2007, the following papers were recognized, (quoted verbatim from the SIGOPS web-page):

- Leslie Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM 21(7):558-565, July 1978.
  Perhaps the first true "distributed systems" paper, it introduced the concept of "causal ordering", which turned out to be useful in many settings. The paper proposed the mechanism it called "logical clocks", but everyone now calls these "Lamport clocks".

- Andrew D. Birrell and Bruce Jay Nelson, Implementing Remote Procedure Calls, ACM Transactions on Computer Systems 2(1):39-59, February 1984.
  *This is *the* paper on RPC, which has become the standard for remote communication in distributed systems and the Internet. The paper does an excellent job laying out the basic model for RPC and the implementation options.*

- J. H. Saltzer, D. P. Reed, and D. D. Clark, End-To-End Arguments in System Design, ACM Transactions on Computer Systems 2(4):277-288, November 1984.
  This paper gave system designers, and especially Internet designers, an elegant framework for making sound decisions. A paper that launched a revolution and, ultimately, a religion.

[1] `http://www.sigops.org/awards/hall-of-fame.html`

- Michael Burrows, Martin Abadi, and Roger Needham, A Logic of Authentication, ACM Transactions on Computer Systems 8(1):18-36, February 1990.
  This paper introduced to the systems community a logic-based notation for authentication protocols to precisely describe certificates, delegations, etc. With this precise description a designer can easily reason whether a protocol is correct or not, and avoid the security flaws that have plagued protocols. "Speaks-for" and "says" are now standard tools for system designers.

- Fred B. Schneider, Implementing Fault-Tolerant Services Using the State Machine Approach: a tutorial, ACM Computing Surveys 22(4):299-319, December 1990.
  The paper that explained how we should think about replication ... a model that turns out to underlie Paxos, Virtual Synchrony, Byzantine replication, and even Transactional 1-Copy Serializability.

Distributed Computing in SOSP and OSDI

Allen Clement
University of Texas at Austin
aclement@cs.utexas.edu

Abstract

SOSP, the *ACM Symposium on Operating Systems Principles*, and OSDI, the *USENIX Symposium no Operating System Design and Implementation*, are "the world's premier forum for researches, developers, programmers, vendors, and teachers of operating systems technology" according to the SOSP home page. While it may seem odd to discuss operating systems conferences in a column dedicated to distributed computing, the proceedings of the last few SOSP and OSDI's have included numerous papers focused on topics more traditionally associated with distributed computing—primarily transactions, Byzantine fault tolerance, and large distributed systems. In this article we highlight papers from the last 3 years of SOSP and OSDI that are especially relevant to the distributed computing community, identifying areas where the distributed computing community was clearly ahead of its time and others where it was not.

1 PODC, ahead of its time...

Transactions are a familiar topic at PODC and DISC, where their existence and importance is taken as a matter of course. The case for renewed interest in transactions, specifically transactional memory, was made in the previous Distributed Computing Column [6, 16, 24]. Not surprisingly, transactions had a substantial presence (12.5% of the accepted papers) at SOSP 2007.

Rossbach et al. [35] explore the challenges of using transactional memory in operating systems. Their premise is that the abstractions of atomic operation and serialized, isolated execution of atomic units greatly simplifies the task of programming parallel systems—which modern systems most certainly are. Unfortunately, operating systems support I/O operations that are not easily included in transactions due to the practical difficulty of rolling back an output that is already visible on the screen. Rossbach et al. solve this problem through the use of *cxspinlocks*, a novel mechanism that dynamically incorporates locks into transactions. Cxspinlocks allow transactions to graciously handle I/O and also provides a solution to the problem of priority inversion in lock based systems.

Sinfonia [4] utilizes *minitransactions* as the primary building block for scalable distributed systems. Minitransactions are not quite transactions, they are "mini" after all, but they capture the abstraction of atomic actions and hide the complexities of concurrent execution and failures from the programmer.

Vandiver et al. [36] present a Byzantine fault tolerant transaction processing database system. While transactions play an important role in the system, the focus is on providing Byzantine fault tolerant (BFT) replication of the transaction execution.

BFT has a long and familiar history in PODC, dating back to 1982 [15] and beyond [34], and a much shorter history in the systems community amidst concerns that the overheads of Byzantine fault tolerant systems are insurmountably high. These concerns are traditionally based on the lower bound of $3f + 1$ replicas required to solve Byzantine consensus [34] and the fundamental impossibility of solving consensus with any failures in an asynchronous system [17].

Castro et al. [9] debunked the myth that the overheads of BFT are prohibitively high at SOSP 1999 with Practical Byzantine Fault Tolerance (PBFT), a replicated state machine based on Byzantine Paxos. Since PBFT demonstrated that BFT could be practical, the SOSP/OSDI community has been increasingly open to papers on the subject. Recent BFT work at SOSP/OSDI can be divided into two broad categories: state machine replication and distributed storage.

PBFT requires $n \geq 3f+1$ PBFT replicas in order to ensure safety in an asynchronous system. When the network is cooperative and the primary is non-faulty, each client request requires $O(n^2)$ messages and 5 message delays to complete—one message delay from the client to the primary and then 4 additional message delays to complete the consensus protocol used to coordinate the replicas. A series of papers from SOSP 2005, OSDI 2006, and SOSP 2007 has attempted to improve on the performance of PBFT by re-evaluating the mechanism used to coordinate the replicas.

The Query/Update (Q/U) protocol [1] replaces the consensus protocol used in PBFT with a quorum based protocol requiring $n \geq 5f + 1$ replicas but that is able to complete client requests in only 2 message delays as long as there are no contending requests in the system. Contention is addressed through client initiated exponential back off. The Hybrid quorum (HQ) protocol [12] presented at OSDI 2006 utilizes quorum techniques requiring $n \geq 3f + 1$ replicas to coordinate client requests in the absence of contention and falls back to PBFT when contention is present. HQ requires 4 message delays to complete client requests. Zyzzyva [25] presented at SOSP 2007 utilizes a fast consensus protocol that requires $n \geq 3f+1$ replicas to ensure safety in an asynchronous system and only 3 message delays to complete client requests in the absence of failures—1 message from the client to the primary and then two more to complete a fast consensus protocol. The authors of Zyzzyva provide an alternate protocol requiring $n \geq 5f + 1$ replicas to ensure safety and only 3 message delays to complete even in the presence of f faults. The key to understanding Zyzzyva lies in understanding the fast consensus protocols developed by the distributed computing community in recent years [14, 31, 18, 27] and realizing that the clients are, adopting Paxos [26] terminology, the *learners* in the system. Zyzzyva demonstrates that batching multiple client requests into a single consensus operation is very valuable, increasing throughput by up to a factor of 6 and reaffirming the observation of Friedman and van Renesse [19]. The ability to batch requests appears to be a strong argument against primary-less quorum systems in replication protocols.

While the evolution from PBFT to Zyzzyva has focused extensively on the computation overheads of coordinating replicas in a distributed system, this computational overhead is not the only challenge faced by distributed storage systems. In storage systems, fault tolerance is achieved by replicating the stored data. Naive replication requires storing n copies of the data. Erasure coding is an attractive technique to reduce the storage and network overheads, but traditional systems employing erasure coding have either increased the server and computational costs or failed to decrease the network overheads. In a striking demonstration of the connection between SOSP and PODC, at SOSP 2007 Hendricks et al. [22] presented a novel BFT storage system based on the homomorphic fingerprinting scheme [23] presented just months before at PODC 2007. While others had addressed the theoretical challenge of reliable distributed storage [2, 20, 32], Hendricks et al. were the first to demonstrate that reliable storage can be done efficiently in practice.

PeerReview [21] describes a Byzantine failure detector that is appropriate for use in distributed systems. One key observation of PeerReview is that faults in distributed systems fall into 2 categories, acts of omission and acts of commission. Acts of omission may prevent liveness while acts of commission are necessary in order to violate safety. PeerReview identifies and assigns blame for acts of commission in the system but does nothing in response to acts of omission. A key challenge in PeerReview is ensuring that acts of commission are correctly identified and that correct participants cannot be falsely accused of faulty behaviors.

2 ... and possibly a little behind

PODC 2006 saw two papers [3, 33] that attempted to integrate selfish and Byzantine behavior. The common motivation for these works is that modern systems are often deployed across multiple administrative domains (MADs) where most users must be treated as potentially selfish actors but the realities of faulty and malicious behavior cannot be avoided. Aiyer et al. [5] actually beat PODC to the punch at SOSP 2005 with the introduction of the Byzantine Altruistic Rational (BAR) model for distributed systems. Under the BAR model, nodes are classified as either **B**yzantine, **A**ltruistic (i.e., correct), or **R**ational. Aiyer et al. use a BAR tolerant consensus based replicated state machine protocol in order to build a peer to peer backup system that is provably correct in the presence of fewer than $\frac{1}{3}$ Byzantine participants even when the non-Byzantine participants are rational rather than correct. OSDI 2006 saw more work in the BAR model with the introduction of BAR Gossip [28]. BAR Gossip is a gossip protocol for streaming live media that is designed to tolerate both Byzantine and rational behaviors by the clients receiving the stream.

While the work in the BAR model [5, 28] and the PODC papers [3, 33] ostensibly address the same problem of integrating Byzantine fault tolerance and game theory, they approach the problem with drastically different motivations and metrics of success. The theory papers naturally strive for a clean model with powerful properties and elegant proofs while the systems papers are driven by implementation concerns and limitations of deployed systems. An upcoming challenge is identifying techniques that are appealing to both system builders and theoreticians.

3 Help needed

As systems conferences, SOSP and OSDI are popular destinations for experience papers describing the design, development, and employment of large scale systems. Dynamo: Amazon's highly available key-value store [13] was presented at SOSP 2007 and Google's Chubby [7] and Bigtable [10] were both presented at OSDI 2006. As experience papers, Dynamo, Chubby, and Bigtable claim little in the way of technical contributions but rather focus on the requirements and challenges of large scale deployments in mission critical environments. Traditionally, academic work has focused on guaranteeing safety always and some form of liveness under the weakest possible conditions. Dynamo, challenges this approach by explicitly settling for weaker safety guarantees in order to achieve better availability and liveness properties: the key store should always be writeable but is allowed to return stale reads. It is also not uncommon for academic systems and protocols to be designed with server faults in mind while assuming that clients are correct. Chubby, a lock service based on Paxos [26], found that client behaviors posed a significant threat to the system's usability, providing evidence that client failures should more consistently considered.

2007 saw an interesting question raised by the SOSP/OSDI community: do we really need $3f+1$ replicas for asynchronous Byzantine consensus? Surprisingly, the answer appears to be no!

Li et al. go "Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems" [30] and introduce *fork* consistency*, a safety constraint weaker than the linearizabilty provided by full state machine replication. A system is fork consistent iff (a) every result accepted by a correct client is based on a history of well formed client requests, (b) correct client i only accepts results based on histories that contain all of i's previous requests, and (c) if two correct clients accept a result for the same request, then the corresponding histories match up until that point. They are able to show a variant of PBFT with $n \geq 3f+1$ nodes that is safe according to traditional definitions as long as at most f replicas are faulty and is fork consistent with up to $2f$ faults. Fork* consistency is based on fork linearizabilty, first used by the SUNDR system [29] at OSDI 2004 and generalized by Cachin et al. [8] at PODC 2007.

Chun et al. [11] claim that consensus can be achieved in a Byzantine setting with only $2f+1$ replicas. The key insight of this paper is the use of a trusted append only memory. As long as the trusted memory is correct, Byzantine replicas are unable to equivocate and send different messages to different replicas.

Chun et al. and Li et al. both challenge fundamental assumptions of the Byzantine fault model, either by relaxing the consistency guarantees or restricting the actions available to Byzantine participants. In doing so they raise important questions about the tradeoffs to be made between consistency and safety semantics, liveness guarantees, and restrictions on the model. These are questions that the distributed computing community is well equipped to explore and answer.

4 Conclusion

This has been a quick overview of papers from recent SOSP and OSDI conferences that should be of interest to the distributed computing community. Hopefully an increased awareness of the work going on in the operating systems community reaffirms the importance of traditional distributed computing work and aids in identifying problems that, if solved, can have a broad impact in the computing world at large.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. 20th SOSP*, Oct. 2005.

[2] I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 226–235, New York, NY, USA, 2004. ACM.

[3] I. Abraham, D. Dolev, R. Gonen, and J. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 53–62, New York, NY, USA, 2006. ACM.

[4] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, New York, NY, USA, 2007. ACM.

[5] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. 20th SOSP*, Oct. 2005.

[6] H. Attiya. Needed: foundations for transactional memory. *SIGACT News*, 39(1):59–61, 2008.

[7] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[8] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 129–138, New York, NY, USA, 2007. ACM.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[11] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 189–204, New York, NY, USA, 2007. ACM.

[12] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. 7th OSDI*, Nov. 2006.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[14] D. Dobre and N. Suri. One-step consensus with zero-degradation. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 137–146, Washington, DC, USA, 2006. IEEE Computer Society.

[15] D. Dolev and R. Reischuk. Bounds on information exchange for byzantine agreement. In *PODC '82: Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 132–140, New York, NY, USA, 1982. ACM.

[16] P. Felber, C. Fetzer, R. Guerraoui, and T. Harris. Transactions are back—but are they the same?: "le retour de martin guerre" (sommersby). *SIGACT News*, 39(1):47–58, 2008.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[18] R. Friedman, A. Mostefaoui, and M. Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Trans. Dependable Secur. Comput.*, 2(1):46–56, 2005.

[19] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocls. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, page 233, Washington, DC, USA, 1997. IEEE Computer Society.

[20] R. Guerraoui, R. R. Levy, and M. Vukolic. Lucky read/write access to robust atomic storage. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 125–136, Washington, DC, USA, 2006. IEEE Computer Society.

[21] A. Haeberlen, P. Kouznetsov, and P. Druschel. Peerreview: practical accountability for distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 175–188, New York, NY, USA, 2007. ACM.

[22] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead byzantine fault-tolerant storage. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 73–86, New York, NY, USA, 2007. ACM.

[23] J. Hendricks, G. R. Ganger, and M. K. Reiter. Verifying distributed erasure-coded data. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 139–146, New York, NY, USA, 2007. ACM.

[24] M. Herlihy and V. Luchangco. Distributed computing and the multicore revolution. *SIGACT News*, 39(1):62–72, 2008.

[25] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[26] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[27] L. Lamport. Lower bounds on asynchronous consensus. In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2003.

[28] H. C. Li, A. Clement, E. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. Bar Gossip. In *Proc. 7th OSDI*, 2006.

[29] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[30] J. Li and D. Mazires. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, April 2007.

[31] J.-P. Martin and L. Alvisi. Fast byzantine consensus. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 402–411, Washington, DC, USA, 2005. IEEE Computer Society.

[32] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325, London, UK, 2002. Springer-Verlag.

[33] T. Moscibroda, S. Schmid, and R. Wattenhofer. When selfish meets evil: byzantine players in a virus inoculation game. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 35–44, New York, NY, USA, 2006. ACM.

[34] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[35] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2007. ACM.

[36] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 59–72, New York, NY, USA, 2007. ACM.

Modularity: a First Class Concept to Address Distributed Systems

Roy Friedman
Computer Science Department, Technion
Haifa 32000, Israel
roy@cs.technion.ac.il

Anne-Marie Kermarrec
INRIA Bretagne-Atlantique
35042 Rennes Cedex, France
Anne-Marie.Kermarrec@inria.fr

Michel Raynal
IRISA, Université de Rennes
35042 Rennes Cedex, France
raynal@irisa.fr

Abstract

Decomposing distributed systems into modules, each with a precise interface and a functional implementation independent specification, is highly effective both from a software engineering point of view and for theoretical purposes. The usefulness of this approach has been demonstrated in the past in several areas of distributed computing. Yet, despite its attractiveness, so far work on peer to peer systems failed to do so. This paper argues in favor of this approach and advocates such a decomposition for peer to peer systems. This allows designers to understand and explain both what a system does and how it does it.

1 Introduction

Designing, building, and reasoning about robust distributed systems are complex tasks. Yet, when considered closely, one can identify several different sources of complexity. Using the terminology introduced by F.P. Brooks in his famous *The Mythical Man-Month* book, some of these difficulties are *essential* while others are *accidental*. That is, some are inherent in these environments, while others can largely be avoided using better abstractions and better separations between levels of abstraction. Simply put, much of the accidental complexity is a result of poor engineering approaches to these problems.

One of the most basic engineering approaches is to modularize a problem domain. That is, the domain is decomposed into *modules*.[1] Each module has a well defined interface, which specifies the methods by which the rest of the world can interact with it. Moreover, the module is assumed to have a functional, implementation-independent, specification. From a software engineering viewpoint, this enables composing modules based on their interface alone. More specifically, each module in such a composition may be written in a different programming language, and provided by an independent vendor. Also, it is possible to replace one implementation of a given module with a different implementation without breaking the integrity of the system. We call this approach the *functional modular approach* to distributed systems.

Interestingly, once we apply the functional modular approach to a given problem, it is also easier to investigate this problem in a formal manner. For example, it is possible to investigate what is the minimal functional specification (or properties) that a module must satisfy in order to enable solving a higher level problem. It is also easier to develop clean, robust, and portable algorithms that have rigorous correctness proofs when the algorithm only needs to concentrate on functional properties of its underlying building blocks (i.e., modules), without worrying how these properties are implemented [18].

We demonstrate this thesis by first examining two established domains, namely distributed shared memory and distributed agreement problems. In both domains, we survey initial models and specifications that have been given in a non-modular approach, and discuss the benefits of their functional modular alternatives.

Then, we turn our attention to peer to peer systems. Despite being a relatively new domain in distributed systems, existing works on peer to peer fail to use such a functional modular approach. As a result, they suffer from the same deficiencies as early works on distributed shared memory and on agreement problems. This tendency is even more pronounced as peer to peer systems were first introduced in the context of specific applications, such a file sharing systems for example. In fact, it took a few years, to acknowledge the potential power of the overlay structure and the routing capabilities and that their applicability could go way beyond file sharing systems. As a result, the implementations are usually described in a monolithic manner and there are no clean formal models describing the precise functionalities of such system. Consequently, it is difficult to understand the precise environmental assumptions required for these systems to work, and there

[1]The concept of a *software component* in software engineering is very similar to a module. The two main differences between these two terms are that components can always be composed as binary units and that components must not have an externally observable state. Therefore, a distributed shared memory "entity" is not a component under this definition. Thus, in this paper we use the term module.

are typically no complete rigorous proofs of correctness. The paper proposes elements to address peer to peer systems according to a functional modular approach.

A first attempt towards that direction was made pretty early on in [7] where common abstractions for peer to peer systems using a 3 tier approach was defined. In this work, the *Key Based Routing* (KBR) abstraction, which allows to route a message to a node based on an abstract key was identified at tier 0. At tier 1, the authors propose the *Distributed Hash Table* (DHT), *Group Anycast and Multicast* (CAST), and *Decentralized Object Location and Routing* (DOLR) abstractions, which rely on the KBR for their implementation. Finally, at tier 2 there are the applications that use one or more of the tier 1 abstractions as services. For the KBR abstraction, they also offer an API. Yet, this work focused on structured peer to peer systems and since then, many other relevant peer to peer systems have been proposed relying on extremely different structure and potentially different routing protocols.

We acknowledge the existence of various peer to peer networks and integrate unstructured peer to peer overlay in our modular description. In that sense, our work can be viewed as complementing the API definitions in [7].

2 Distributed shared memory

Initial works on Distributed Shared Memory (DSM), like Release Consistency [12] and Java Consistency [14], have mixed implementation details with specifications. This resulted in specifications that were extremely difficult to understand and prove. In the case of Java consistency, as has been shown in [20], several commercial compilers by major vendors did not implement correctly the memory model of Java according to its initially unclear specification. On the other hand, the approaches taken in works on Sequential Consistency, Linearizability, Hybrid Consistency, Causal Ordering (see Appendix A), and the non-operational definition of Java Consistency [13], have proposed to make a clean separation between implementation details and functionality.

In these approaches, each node can be seen as made up of two parts: the local application process and a DSM abstraction that offers it appropriate access primitives, such as `read` and `write` in the case of read/write objects, `dequeue` and `enqueue` in the case of queue objects, etc. Then, a formal functional specification of the DSM abstraction is given with respect to its interface signature. That is, the specification only defines allowed collections of sequences (one for each process) of method invocations and their returned values. In particular, this specification does not say anything about the implementation of the DSM module. This way, various consistency conditions simply differ in the restrictions they impose on such sequences of method invocations and returned values. Once this definition framework is established, for any consistency condition, it is possible to develop programs and prove their correctness only based on the functional specification of the condition. Similarly, it is possible to prove lower bounds on the requirements from any possible implementation of a given condition, as was done (e.g., in [3]).

The DSM abstraction can in turn be decomposed into several modules, which include a DSM protocol, a local storage module and a network interface, as illustrated in Figure 1 (the local storage module is usually used as a local cache endowed with a persistence property, the shared virtual memory being distributed across the local storages).

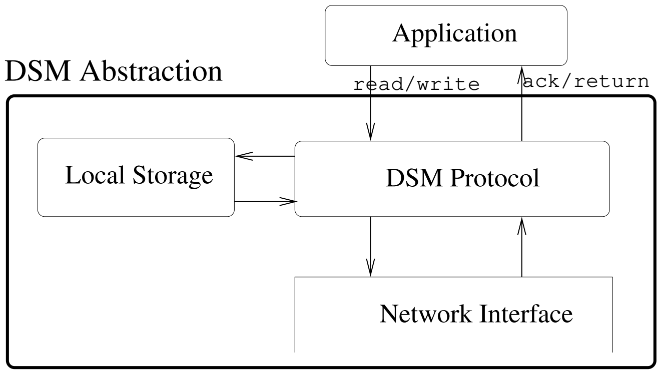


Figure 1: A typical modular view (for a node) of DSM.

The important thing is that once the functional modular approach was taken to this problem, it facilitated understanding exactly the semantics provided for the application by each consistency model [2]. Moreover, it allowed for clear comparisons between different consistency conditions, and for rigorous correctness proofs of both applications and implementations. In particular, it enabled designing applications and proving their correctness with respect to a given consistency condition without relying on specific implementation details or assumptions. To a large extent, it also simplifies implementing a consistency condition, since it highlights what is needed and why.

3 Agreement problems in semi-synchronous distributed environments

Agreement is a fundamental issue in distributed computing. In distributed systems, processes often collaborate in order to achieve a common goal. This usually involves reaching some level of agreement, e.g., on the next state of the system, on the next course of action, on the allocation of a resource, etc.

A precondition to designing and reasoning about solving problems in distributed systems is having an adequate model for it. Initially, two extreme models were defined for distributed systems: *synchronous* and *asynchronous*. In the former, all actions occur within a known deadline and there is a shared global clock available to all processes. In the latter, processes have no access to any global clock and there is no bound of the latency of events.

Yet, with the exception of specially crafted real-time systems, most standard distributed systems do not continuously and dependably exhibit the strict timing assumptions of the synchronous model. On the other hand, it was shown that basic agreement problems cannot be solved in purely asynchronous environments [10]. Moreover, it was observed that realistic standard-based distributed systems typically obey some level of synchrony during large fractions of their lifetime. This lead to the definition of models with explicit timing assumptions (e.g., the *timed-asynchronous* [6] model, or the synchrony models described in [8]). However, because these models have explicit timing assumptions in them, it is harder to develop generic portable protocols, and it is hard to formally investigate and state the minimal system requirements for solving agreement problems (see Appendix B).

A cleaner modular solution was proposed by Chandra and Toueg, who introduced in their seminal work the notion of a *failure detector* [5]. That is, Chandra and Toueg observed that the difficulty of reaching agreement in asynchronous distributed systems stems from the fact that it is not possible to distinguish a failed process from a slow one in these environments. Thus, they proposed to enrich the environment with a failure detector module. The interface of a failure detector includes a method which returns a list of suspected (alternatively, a list of trusted) processes. Given this interface, it is possible to define the functional, implementation independent, semantics of the failure detector module with respect to the values returned when invoking its method. In particular, Chandra and Toueg defined several types of *completeness* and *accuracy* properties. Completeness defines the degree to which failed processes should be included in the list of suspected processes while accuracy defines limitations on falsely suspecting alive processes. As an example, a failure detector of the class denoted $\Diamond\mathcal{S}$ eventually suspects all the crashed processes (completeness), and ensures that after some unknown but finite time there is a correct process that is no longer suspected (eventual weak accuracy). The failure detector approach enabled identifying precisely what are the minimal levels of completeness and accuracy that are required to solve various agreement problems (e.g., $\Diamond\mathcal{S}$ has been shown to be the weakest class of failure detector modules that allow solving consensus despite the net effect of asynchrony and crash failures). Moreover, the failure detector approach was used to develop robust portable protocols, whose correctness depends only on the functional behavior of the failure detector module and not on its implementation or the other environment assumptions.

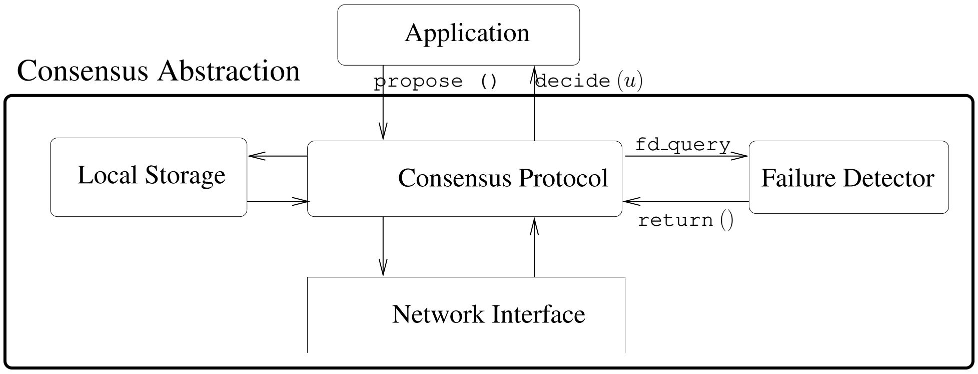


Figure 2: A modular view (for a node) of an asynchronous system enriched with a failure detector.

Traditionally, it was thought that the only way to implement a failure detector module is by assuming that the underlying system obeys some minimal timing assumptions, at least most of the time. With this assumption, it is possible to run some sort of a *heartbeat* based protocol in order to detect failures.

Interestingly, it turned out that the class of failure detector modules $\Diamond\mathcal{S}$, which enables solving Consensus in otherwise asynchronous distributed settings, can be also implemented in timeless environments [19]. The only requirement is that the order in which each process receives replies to queries it sends all other processes obeys some minimal constraints. This highlights the thesis advocated in this paper, since without the functional modular approach, it would have been very difficult to obtain this result. Moreover, all protocols that were designed to solve Consensus based on $\Diamond\mathcal{S}$ work correctly in both environments, each with its corresponding implementation of a $\Diamond\mathcal{S}$ module. This again exhibits the benefits of the functional modular approach.

4 Peer to peer systems

Peer to peer computing has received increasing attention over the past years and remains a relatively recent area of distributed computing compared to the distributed systems described so far. The main characteristic of peer to peer systems over previous distributed systems is to be serverless. More precisely, end users (nodes) potentially act both as client and server. They communicate directly with each other, and provide the system with services in a collaborative manner, rather than relying on dedicated servers for this. The promise of peer to peer approaches is scalability and long term survivability. Scalability is achieved by enabling the system to operate while each node only needs to know about a small fraction of the whole system. To this end, an overlay network, connecting nodes in a logical manner, is built on top of a physical network. Distributed Hash Tables-based implementations, such as Pastry or Chord, have dominated at first, e.g., [21, 23, 26] (see Appendix C). Since then, many other approaches, such as gossip-based unstructured overlay networks have been proposed. Peer to peer systems are designed to cope with dynamics and get automatically reorganized upon node joins and departures.

Yet, despite being an emergent domain, existing research on peer to peer still suffers from the same unsatisfactory engineering practices of initial works in other distributed computing domains.[2] There is a lack of clear separation between levels of abstractions, and clear specifications of such systems. More specifically, no clear specification of the exact conditions under which the system will behave as promised is provided.[3]

This paper advocates the use of the functional modular decomposition for peer to peer systems. The proposed architecture may not be the ultimate one, but we use it as a proof of concept. We show that by taking the functional modular approach, we can specify a generic peer to peer system. The different modules of this generic architecture can then be instantiated, each with a specific implementation. The presentation that follows is voluntarily informal. Its aim is only to show how peer to peer systems can benefit from the modular approach.

4.1 Problem statement

We assume a system operating in a distributed environment composed of a finite but unbounded and changing set of processes. We refer to each change in the set of processes as a *configuration change*. Peer to peer systems can be used for many purposes. Yet, for the sake of simplicity and for consistency with the previous sections (Section 2 and Section 3), we consider a system whose goal is to implement a *semi-reliable unified storage* abstraction. That is, we call a collection of sequences of read and write operations, each of these sequences executed by a single process, a

[2]This is, of course, with the exception of [7], as discussed in the Introduction.

[3]We would like to emphasize that while the actual implementations of these systems might be written in object oriented programming languages, their design, at least as it appears in research papers describing them, are not modular.

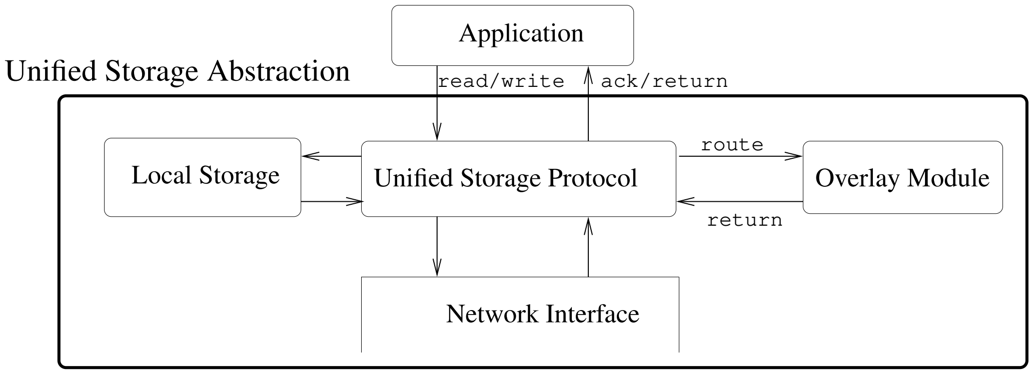


Figure 3: Modular view (for a node) of a peer to peer system.

partial execution. We call an *uninterrupted partial execution* a partial execution during which there are no failures or configuration changes. We then require that for each uninterrupted partial execution σ of the system, it is possible to find a total order ϕ that extends the real time ordering of all operations in σ such that each read operation for a given object o returns the value written by the last write to o that precedes it in ϕ. This is similar to the notion of *linearizability* (see Appendix A) restricted to an uninterrupted partial execution.

Notice that this specification is weak in the sense that it does not require anything w.r.t. operations that occur during periods in which the system is unstable. Finding a meaningful formal specification that captures the entire lifespan of the system, and yet is simple to grasp, is left as an open challenge to the readers of this article.

The reason we are only interested in semi-reliable storage is that most peer to peer systems implement a location service. The goal of a location service is to allow finding an object. Thus, a possible way to implement such a location service is to have each object write its location in the semi-reliable storage. Then, for lookup, a node tries to read the location of the object from the semi-reliable storage until it gets an answer.

4.2 Architecture

Figure 3 illustrates a modular view of a peer to peer system. In such a system, all nodes are symmetric. This figure has the same structure as the two previous ones. The only difference is that, on each node, the corresponding application process accesses a unified storage abstraction (instead of a DSM or an agreement abstraction). The unified storage abstraction is made up of four parts: a *unified storage* protocol, an *overlay* module (see below), a *local storage* module, and a *network interface*.

The unified storage module consults the overlay module to perform read/query and write/update request ordered by the application. To this end, the overlay module exposes a single method called `route`, which accepts an object identifier and the current process id and returns the next process(es) id(s) to whom the message should be routed to. Thus, each time the application invokes a read or a write, the unified storage module invokes the `route` method and forwards the message to the corresponding process(es). This continues until the `route` method returns the same process id as the calling process. The latter indicates to the unified storage module that it is the one responsible for storing the object. For this, the unified storage module uses the local storage module. The corresponding protocol is summarized in Figure 4 (with ts initialized to 0 before the execution begins). Note that for clarity of presentation, in Figure 4 we consider that only one process id is returned at each step.

```
Upon Write(o, v) from the application do
    ts[o] ← ts[o] + 1;
    p_l ← overlay.route(p_i,o);
    send (WRITE,o,v,(ts[o], i)) to p_l;
    return

Upon Read(o) from the application do
    p_l ← overlay.route(p_i,o);
    send (QUERY,o,p_i) to p_l;
    wait until (RESPONSE,o,v,(t, j)) is received;
    ts[o] ← max(ts[o],t);
    return v

Upon receiving (QUERY,o,p_k) from the network do
    next ← overlay.route(p_i,o);
    if next = p_i then send (RESPONSE,o,v,(ts[o], j)) to p_k;
                       % note that p_k is the originator of (QUERY,o,p_k).
                       % So p_i directly sends the response to p_k
                       % (ts[o], j) is the timestamp that p_i associated with o
                       % if o does not exist locally, its local timestamp
                       % is 0 and the value is the default initial value
               else  send (QUERY,o,p_k) to next
    endif

Upon receiving (WRITE,o,v,(ts, j)) from the network do
    next ← overlay.route(p_i,o);
    if next = p_i then if (ts, j) is larger than the timestamp of p_i's copy of o
                            then update the value of the local copy of o to v
                            update the timestamp of the local copy of o to (ts, j)
                            % if o does not exist locally, this initializes its copy
                       endif
              else  send (WRITE,o,v,ts) to next
    endif
```

Figure 4: A generic peer to peer protocol (code for node p_i).

4.3 The overlay module

In order to define the properties of the overlay module, we first define the following concepts: A *targeted invocation sequence* is a sequence of invocations of the `route` method such that (a) all

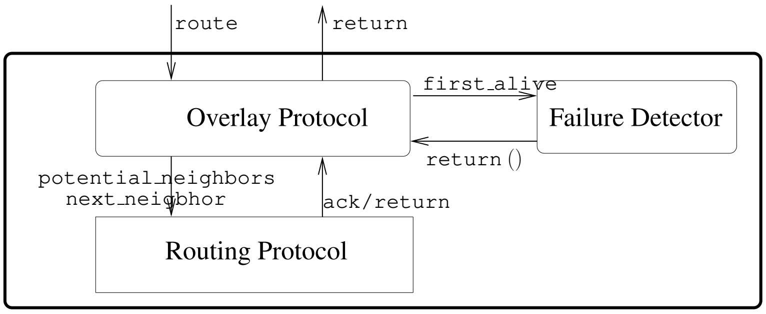


Figure 5: Decomposing the overlay module.

invocations are called with the same object identifier, and (b) the process id passed to the $(i+1)^{\text{th}}$ invocation is the one returned by the i^{th} invocation in the sequence. We say that a targeted invocation sequence *converges* if it includes two consecutive invocations of the `route` method that return the same process id. If the sequence converges, we call the first process id that appears in such a pair of consecutive invocations in this sequence the *converged process*. Finally, for a given execution of the system, we say that two targeted invocation sequences are *undisturbed* if there are no failures and no configuration changes during the entire interval from the earliest of these invocations until the time at which the latest of them returns.

Based on these definitions, the `route` method of the overlay module must satisfy the following properties (these requirements are the functional specification of the overlay module):

- Route Convergence: There exists a function $f(n)$ such that any targeted invocation sequence of length at least $f(n)$ converges.
- Route Consistency: Any two static undisturbed targeted invocation sequences in which the `route` method is invoked with the same object id converge to the same converged process[4]. Note that depending on the application and the system considered, the invocation of the route method may return several processes.

Periodically do
  candidates $\leftarrow$ *RP*.`potential-neighbors`(p_i);
  empty *finger*;
  foreach entry k in *candidates* **do**
    finger[k] $\leftarrow$ *FD*.`First-alive`(*candidates*[k])
  enddo

Upon `route`(p_i,*obj*) **do**
  return (*RP*.`next-neighbor`(*finger*,*obj*))

Figure 6: Overlay module.

Therefore the overlay module should specify *(i)* the structure of peer to peer overlay network and *(ii)* the routing protocol. Internally, the overlay module is based on three sub-modules, as

[4]It is possible to refine this definition to limit what can happen even during configuration changes and failures.

illustrated in Figure 5: the overlay protocol, the routing module, and a failure detector module. The overlay protocol is aided by the other two modules. The aim of this "high level" description is to show the benefit of the modular decomposition approach. When mapping various peer to peer systems to the model we have introduced, many of them differ in the function $f(n)$ of the definition of Route Convergence (or the *overlay network diameter*), and in the size of the list returned by the `potential-neighbors` method (or the *node degree* in the overlay). Our decomposition allows investigating lower bounds on obtaining these properties, as well as comparing known protocols based on a common ground.

In the next section, we provide more details on each of these components. More specifically, we provide examples of several potential implementations of possible overlay and routing modules. Obviously, these two modules are strongly related as for a given overlay structure, a specific routing protocol might be the most relevant. Likewise the choice of a specific overlay structure or routing protocol may be dependent of the application targeted. Yet, this decomposition both covers existing approaches and leaves space for new ones.

4.3.1 Overlay protocol

This protocol specifies the way the peer to peer overlay network is structured. Existing peer to peer overlay networks mostly differ in the structure they impose to the logical overlay topology. At one end of the spectrum lie the *structured overlay networks*, while at the other end, we find fully *unstructured overlay networks*. Many possible intermediary structures might be considered. Usually, the more structured a peer to peer overlay is, the more efficient the routing protocol becomes, but it offers less flexibility or expressiveness.

The overlay structure is fully defined by the local *knowledge* of the system, precisely provided by the set of neighbors (Id and IP address) that each node maintains. The overlay protocol is fully specified by the set of neighbors maintained at each node, specifying the structure of the peer to peer overlay network, the join and the maintenance protocols.

Structured overlay networks In structured peer to peer networks, the peers are organized in a pre-defined structure such as tree, a hypercube, a ring, etc. To this end, peers get assigned coordinates in an id space, usually in a random and uniform way. This ensures that the ids are uniformly distributed in that space. This assumption, by avoiding skewed distribution of ids, enables to evenly balance the load between peers in the systems. The *set of neighbors* maintained at each node is strongly dependent on the structure of the overlay. In structured peer to peer overlays, strict constraints are imposed on the choice of neighbors, usually through the identifier. For example in a *ring*-based overlay, each node should know about its predecessor and successor in the one dimension identifier space. In CAN [22], each node should know about its predecessor and successor along each dimension in a d-dimensional coordinate space. In Pastry, node-IDs are r digit numbers of base 2^b [21]. Each Pastry node maintains two sets of neighbors namely a *leaf set* and a *routing table*. The leaf set contains a fixed number of entries whose node IDs are numerically closer to the local node ID. In Pastry routing tables, the entry in the cell $[i, j]$ has the first $i-1$ digits of its node ID same as the local node-ID and the i^{th} digit as j.

The *join protocol* specifies the actions taken upon join. The aggressiveness of the join protocol has a strong impact on the time required upon join to converge towards the targeted structure. Node joins are usually handled by the basic routing mechanism. When a new node joins the overlay, it uses the routing protocol to find the closest node to its own ID and inserts itself adjacent to that node in the overlay. Some additional operations may be required to make the set of neighbors converged more quickly.

Finally, the maintenance protocol specifies the way the structure is maintained, more specifically, this protocol enables to cope with system dynamics. For instance, Pastry implements a lightweight maintenance protocol when periodically, each node exchanges a line, chosen randomly, of its routing table with a neighbor, chosen randomly, of this specific line. A specific repair protocol is implemented when a failure is detected during the routing operation.

Unstructured overlay networks In unstructured overlays, each node maintains a set of c arbitrary neighbors. Random graph-like topologies have received an increasing interest as they provide a sound basis to implement reliable dissemination for example. Several protocols have been proposed to build and maintain unstructured overlay networks [1, 9, 24]. In that space, gossip-based protocols have received an increasing interest. In such protocols each node periodically exchanges information with another node randomly chosen from its neighbor set. This periodic exchange between peers spreads the information in the system in an epidemic manner. Gossip-based algorithms have been used to create random-like topologies [16]. Several instances exist and differ by the way they select the peer to communicate with, the list of peers they exchange and the way the compute their new set of neighbors. The join procedure is extremely simple and consists for a new node to contact a peer in the network and bootstrap from its set of neighbors. The periodic maintenance protocol ensures that the neighbor set converge to a random set of neighbors.

Some protocols such as SCAMP [11] provides each peer with a set of random neighbors. The originality of SCAMP is to provide each node with $O(log(n))$ neighbors without any node knowing explicitly the size of the network. SCAMP does not implement specific maintenance (the (un)structure is achieved through the join protocol only).

Weakly structured overlay networks In small-world based topologies, each node in a mesh, knows its *closest*[5] neighbors and has additional shortcuts in the graph. The asymptotic routing performance depends on the way shortcuts are chosen (random [25] or following a specific distribution such as the d-harmonic distribution [17]). Small-world topologies with random shortcuts can be achieved using a simple random peer sampling protocol and a specific clustering protocol [15]. Small-world topologies with a harmonic distribution of shortcuts can be implemented by a biased peer sampling protocol and a similar clustering protocol [4]. Interestingly, these protocols achieves the targeted structure in a few cycles. Therefore a simple join operation is usually not enough to achieve such a topology and the maintenance protocol is compulsory.

[5]The proximity metric may be application-dependent.

4.3.2 The routing protocol

The routing module has two methods, namely, `potential-neighbors` and `next-neighbor`. The `potential-neighbors` accepts a process identifier and returns an array of ordered lists of processes. The `next-neighbor` accepts as parameters an object identifier and a list of processes and returns the id of one or several of the processes appearing in its input list.

While the overlay protocol enables to build a peer to peer overlay following a given structure, the routing protocol provides the means to navigate such an overlay. The main metric considered to evaluate a routing protocol is the complexity and the exhaustiveness of the result. While many routing protocols may be considered, the most used in the area remains the *greedy* protocol. In such a routing protocol, each routing step gets the message closer to the destination. In the context of unstructured overlay though, a greedy protocol achieves a routing convergence with a $O(n)$ complexity (where n is the system size). In order to improve on latency, flooding or restricted flooding is possible.

Structured overlay networks Many structured overlays rely on a *key-based routing* protocol. Chord routing protocol is provided in Annex C. In CAN, the message progresses along a route in a d-dimensional space along one of the dimension in a greedy manner. In Pastry, the messages are routed through nodes with increasing prefix matching the destination.

Unstructured overlay networks Greedy routing in unstructured overlays leads to a linear complexity in the size of the network. Potentially all nodes should be visited to achieve routing convergence. Flooding protocols are the most used protocols and consists at each peer in flooding the route message to all its neighbors, until the destination is reached. The restricted flooding protocol, assigns a TTL (time to live) to each lookup operation. The TTL is decremented at each hop and the route procedure terminates once the TTL reaches 0. Route convergence might not be guaranteed in such cases. Note that routing to a fixed id is definitely not the most efficient operation in such networks. However, unstructured overlay assorted with flooding or restricted flooding protocols are used for more expressive lookup operations such as range or keyword-based queries. Another option is to perform random walks with a bounded TTL. Yet in order for random walks to have a good chance of find a given data item within reasonable time, there has to be a way to bias them, as is done, e.g., in the GIA system.

4.4 The failure detector module

The failure detector module supports one method (denoted `first-alive` in Figure 5) that accepts an ordered list of processes and returns the id of the first k processes on the list that are alive. The implementation of the overlay based on the routing and failure detector modules appears in Figure 6. (see also Appendix D.)

5 Conclusion

In summary, the great benefits of the functional modular approach are that, after decomposing the system this way, it is easy to understand both what the system does, and how it does it. This is a direct consequence of the clean separation between the different abstraction layers. This permits one to define precisely what is the functional specification of each module, and to come up with a generic protocol that is based only on the functional properties of the underlying modules. This also enables further investigation into what are the minimal required environment assumptions, and to write correctness proofs for the protocols. It also enables better comparisons between systems, as it highlights the essence of their differences.

In the context of peer-to-peer systems, this paper has taken a first step towards such a specification. Yet, our specification is limited to storage oriented systems and lookup services. Also, our specification does not cover the behavior of the system during periods of instability (churn). Extending our specification and capturing other types of peer-to-peer systems is left for future work.

References

[1] `http://www.gnutella.com/`

[2] Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42(1): 124-142, 1995.

[3] Attiya H. and Welch J.L., Sequential consistency versus linearizability. *ACM Transactions On Computer Systems*, 12(2):91-122, 1994.

[4] Bonnet F., Kermarrec A.-M. and Raynal M., Small-world networks: from theoretical bounds to practical systems. *Proc. 11th Int'l Conference On Principles Of Distributed Systems (OPODIS'07)*, Springer-Verlag LNCS 4878, pp. 372-385, 2007.

[5] Chandra T.D. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267, 1996.

[6] Cristian F. and Fetzer C., The timed asynchronous system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642-657, 1999.

[7] Dabek F., Zhao B., Druschel P., Kubiatowicz J., and Stoica I., Towards a common API for structured peer to peer overlays. *Proc. 2nd Int'l Workshop on peer to peer Systems (IPTPS '03)*, Berkeley, CA, 2003.

[8] Dwork C., Lynch N. and Stockmeyer L.J., Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2): 288-323, 1988.

[9] Eugster P., Handurukande S., Guerraoui R., Kermarrec A.-M., and Kouznetsov P., Lightweight probabilistic broadcast, *ACM Transactions on Computer Systems*, 21(4):341-374, 2003.

[10] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382, 1985.

[11] Ganesh A.J., Kermarrec A.-M., and Massoulié L., Peer to peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139-149, 2003.

[12] Gharachorloo K., Lenoski D., Laudon J., Gibbons P., Gupta A. and Hennessy J., Memory consistency and event ordering in scalable shared-memory multiprocessor. *Proc. 17th Int'l Symposium on Computer Architecture (ISCA'90)*, pp. 15-90, 1990.

[13] Gontmake A. and Schuster A., Non-operational characterizations for Java memory model. *ACM Transactions On Computer Systems (TOCS)*, 18(4):333-386, 2000.

[14] Gosling J., Joy G. and Steele G., *The Java Language Specification*, Addison-Wesley, 1996.

[15] Jelasity M. and Babaoglu O., T-Man: Gossip-Based Overlay Topology Management *Engineering Self-Organising Systems*,1(15), 2005.

[16] Jelasity M., Voulgaris S., Guerraoui R., Kermarrec A.-M, and van Steen M. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3), 2007.

[17] Kleinberg J. The small-world phenomenon: An algorithmic perspective. *Proc. 32nd ACM Symposium on Theory of Computing (STOC'00)*, ACM press, pp. 163-170, 2000.

[18] Lamport, L. Composition: A Way to Make Proofs Harder. *Revised Lectures from the Int'l Symposium on Compositionality: The Significant Difference (COMPOS)*, Springer-Verlag, LNCS #1998, pp. 402-423, 1998.

[19] Mostefaoui A., Mourgaya E., and Raynal M., Asynchronous implementation of failure detectors. *Proc. Int'l IEEE Conference on Dependable Systems and Networks (DSN'03)*, IEEE Computer Press, pp. 351-360, 2003.

[20] Pugh W., Fixing the Java memory model. *Proc. ACM Conference on Java Grande (JAVA'99)*, ACM Press, pp. 89-98, 1999.

[21] Rowstron A. and Druschel P., Pastry: scalable, distributed object location and routing for large scale peer to peer systems, *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[22] Ratnasamy S., Francis P., Handley M., Karp R., and Shenker S. A Scalable Content-Addressable Network. *Proc. of the Conference of the ACM Special Interest Group on Data Communications (SIGCOMM'01)*, ACM Press, pp. 161-172, 2001.

[23] Stoica I., Morris R., Liben-Nowell D., Karger D., Kaashoek M.F., Dabek F. and Balakrishnan H., Chord: A scalable peer to peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17-32, 2003.

[24] Voulgaris S., Gavidia D. and van Steen M., CYCLON: inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197-217, 2005.

[25] Watts D.J. and Stogatz S.H. Collective dynamics of small-world networks. *Nature*, 393, 1998.

[26] Zhao B.Y., Kubiatowicz J.D. and Joseph A.D., Tapestry: an infrastructure for fault-tolerant wide-area location and routing, *UCB/CSD-01-1141, Computer Science Department, U.C. Berkeley*, 2001.

A Consistency criteria for DSM

The definition of a consistency criterion is crucial for the correctness of a multiprocess program. Basically, a consistency criterion defines which value has to be returned when a read operation on a shared object is invoked by a process. The strongest (i.e., most constraining) consistency criterion is *atomic consistency* [A7] (also called *linearizability* when we consider objects more sophisticated than simple read/write shared variables [A4]). It states that a read returns the value written by the last preceding write, "last" referring to real-time occurrence order (concurrent writes being ordered). *Causal consistency* [A2] is a weaker criterion stating that a read does not get an overwritten value. Causal consistency allows concurrent writes; consequently, it is possible that concurrent read operations on the same object get different values (this occurs when those values have been produced by concurrent writes). Causal consistency is encountered in some cooperative applications. Other consistency criteria (weaker than causal consistency) have also been proposed [A1,A3].

Sequential consistency [A5] is a criterion that lies between atomic consistency and causal consistency. Informally it states that a multiprocess program executes correctly if its results could have been produced by executing that program on a single processor system. This means that an execution is correct if we can totally order its operations in such a way that (1) the order of operations in each process is preserved, and (2) each read gets the last previously written value, "last" referring here to the total order. The difference between atomic consistency and sequential consistency lies in the meaning of the word "last". This word refers to real-time when we consider atomic consistency, while it refers to a logical time notion when we consider sequential consistency (namely the logical time defined by the total order). The main difference between sequential consistency and causal consistency lies in the fact that (as atomic consistency) sequential consistency orders all write operations, while causal consistency does not require to order concurrent writes.

The figures 7 and 8 show the difference between atomic consistency and sequential consistency. (Solid arrows denote "process" order, while dotted arrows denote "read-from" order.) Let us consider Figure 7. There are two processes, p_1 and p_2 whose executions are as follows (where $r_i(x)a$ means "p_i reads x and obtains the value a", and $w_i(x)a$ means "p_i writes a in x"): p_1 first writes 0 into x, then reads x and obtains the value 1, and finally reads again x and obtains the value 2; p_2 writes twice in x, first the value 1, then the value 2.

For this execution to be atomically consistent we must be able to totally order all its operations in such a way that (1) the real-time order on operations is respected (e.g., as $w_1(x)0$ is terminated when $w_2(x)1$ starts it has to appear before in the total order), (2) concurrent operations can be ordered in any way, and (3) each read operation obtains the value of the last preceding write ("last" with respect to this total order). The consistent total order $w_1(x)0$, $w_2(x)1$, $r_1(x)1$, $w_2(x)2$, $r_2(x)2$

can be associated with the execution of Figure 7. Hence, it it atomically consistent.

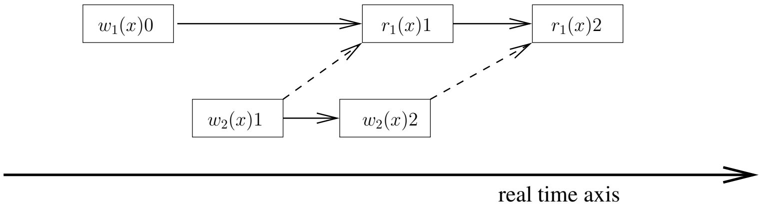


Figure 7: An atomically consistent execution.

Differently, the execution described in Figure 8 is only sequentially consistent. It is possible to totally order all its operations while respecting only the order inside each process and the read-from relation: $w_2(x)1$, $w_2(y)2$, $r_2(x)1$, $w_1(x)0$, $r_1(y)2$, $r_1(x)0$. As the reader can check, this execution is not atomically consistent as it is not possible to totally order its operations while respecting the realtime order of non-overlapping operations.

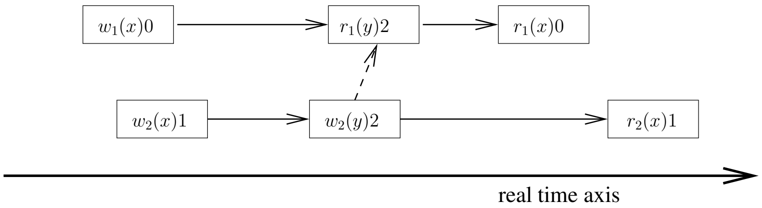


Figure 8: A sequentially consistent execution.

Both atomic consistency and sequential consistency require a sequential "witness" execution. In each case a read obtains the last written value in this sequential execution. As already indicated, "last" is with respect to real-time for atomic consistency, while it is with respect to some logical time for sequential consistency. That is why these consistency criteria demand different protocols when one wants to manage cached values in distributed systems [A6].

[A1] Adve S.V. and Garachorloo K., Shared Memory Models: a Tutorial. *IEEE Computer*, 29(12):66-77, 1997.
[A2] Ahamad M., Hutto P.W., Neiger G., Burns J.E. and Kohli P., Causal memory: Definitions, Implementations and Programming. *Distributed Computing*, 9:37-49, 1995.
[A3] Attiya H. and Friedman R., A Correctness Condition for High-Performance Multiprocessors. *SIAM Journal of Computing*, 27(2):1637-1670, 1998.
[A4] Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.
[A5] Lamport L., How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979.
[A6] Li K. and Hudak P., Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321-359, 1989.
[A7] Misra J., Axioms for Memory Access in Asynchronous Hardware Systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153, 1986.

B Why consensus is important in distributed systems

Considering a message-passing asynchronous distributed system where nodes can crash, the *atomic broadcast* problem states that (1) all the nodes that do not crash must deliver the same set of messages in the same order, and (2) the nodes that crash deliver a prefix of this sequence of messages. This sequence includes only messages broadcast by the nodes, and must include at least the messages broadcast by the correct nodes.

Chandra and Toueg have shown that the atomic broadcast problem and the consensus problem are equivalent in the sense that any of them can be solved as soon as we are given a protocol solving the other. Figure 9 shows how such a transformation works. When a node wants to broadcast a message m it invokes the primitive $TO\_Broadcast(m)$ which simply consists in sending m to all the nodes using the underlying primitive $UR\_Broadcast(m)$ (a uniform reliable broadcast primitive [B4]).

The messages received by a node i are stored in a set $UR\_Delivered_i$. Then the nodes use consecutive consensus instances. During each consensus instance, each node "proposes" a delivery order for the messages it has received (and not yet delivered) from the other nodes. As each consensus instance imposes the same batch of messages to all the nodes, and all the nodes that have not crashed execute the same sequence of consensus instances, they deliver the messages in the same order. These messages are stored in a local queue by each node i, that can then deliver the same sequence of messages (as determined by the sequence of message batches) to the application process located on this node. This shows that atomic broadcast is at the same time a communication problem (messages have to be delivered) and an agreement problem (in the same order). More on this can be found in [B3,B5].

Also, several file systems and real-world replication modules are based on consensus protocols, e.g., [B1] and [B2].

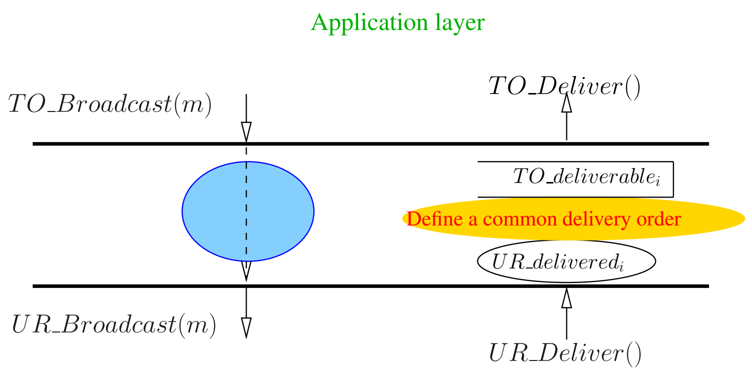


Figure 9: From consensus to atomic broadcast.

[B1] Burrows M., The Chubby Lock Service for Loosely-Coupled Distributed Systems. *7th Symposium on Operating System Design and Implementation (OSDI)*, 2006.
[B2] Chandra T.D., Griesemer R., and Redstone J., Paxos Made Live - An Engineering Perspective.

26th ACM Symposium on Principles of Distributed Computing, pp. 398-407, 2007.
[B3] Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *JACM*, 43(2):225-267, 1996.
[B4] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.
[B5] Mostéfaoui A. and Raynal M., Low-Cost Consensus-Based Atomic Broadcast. *7th IEEE Pacific Rim Int. Symposium on Dependable Computing (PRDC'2000)*, IEEE Computer Society Press, pp. 45-52, 2000.

C A look at Chord

Chord [C2] is one of the first peer to peer systems to use distributed hash tables for data location. The main idea of Chord is to assign to each node and data item an hashed identifier in the range $[0 - 2^{m-1}]$ for some value of m. Thus, each hashed identifier is m bits long. For example, if m is 128, which is often the case, then each identifier is of length 128 bit. Choosing such a value for m ensures that the chances of collisions are negligible and therefore we can assume that indeed no two data items are assigned the same identifier.

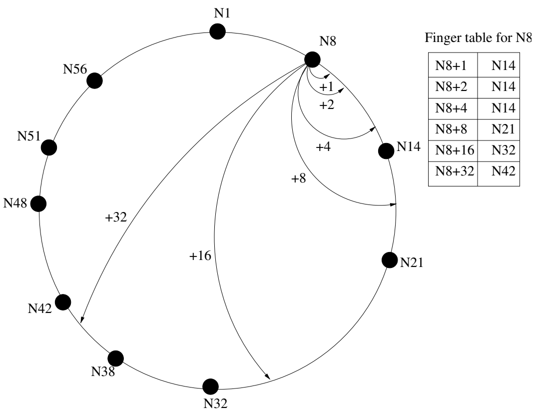


Figure 10: The Chord peer to peer protocol assuming $m = 6$ (from [C2]).

Next, Chord logically places all hashed identifiers, for both nodes and data items, on a logical ring, as illustrated in Figure 10. The goal of the system is to have each node pointing to only $\log n$ other nodes, and have an overlay diameter of $\log n$ as well. Thus, ideally we would like a node that is assigned an identifier i to maintain pointers to all other nodes whose identifiers are $i + 2^j \mod 2^m$ for $j \in [0, 1, \dots, \log n]$ (so the last node is i itself). However, given that the actual number of nodes is much smaller than 2^{128}, then most potential identifiers are not assigned to any node. Thus, instead of keeping a pointer to a node k, node i keeps a pointer to node $\texttt{successor}(k)$, i.e., to the first node clockwise at k or afterwards in the logical ring. This list of nodes that i points to is referred to as the *finger* table of i.

Finding a data item with a hashed identifier d is done by having each node with identifier i propagating the request to the first entry in the finger table of i such that the identifier k' stored there is the largest node that precedes (or equals) d in modulo 2^m.

Placing Chord in our terminology, the `potential_neighbors` method of the DHT would return an array such that each entry k of the array consists of the range of identifiers $[i+2^k, \ldots, 2^{k+m} \bmod 2^m]$. The `first_alive` method of the failure detector would return the first identifier that is populated by an alive node. This serves as the entries in the finger table. Finally, the `next_neighbor` method of the DHT would return the first entry in the finger table that recedes (or equals) d in modulo 2^m.

Note that to adapt to a different peer to peer protocol (such as Pastry [C1] or Tapestry [C3]) all that is needed is to modify the values returned by the `potential_neighbors` and the `next_neighbor` methods.

[C1] Rowstron A. and Druschel P., Pastry: Scalable, Distributed Object Location and Routing for Large Scale peer to peer Systems, *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.
[C2] Stoica I., Morris R., Liben-Nowell D., Karger D., Kaashoek M.F., Dabek F. and Balakrishnan H., Chord: A Scalable peer to peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17-32, 2003.
[C3] Zhao B.Y., Kubiatowicz J.D. and Joseph A.D., Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing, *UCB/CSD-01-1141, Computer Science Department, U.C. Berkeley*, April 2001.

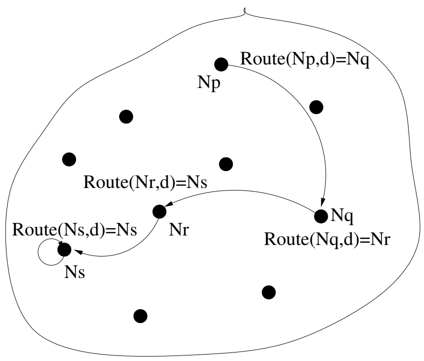


Figure 11: The `route` invocation process in the generic case.

D Looking for a data item in a peer to peer overlay

Following the description in Section 4, we demonstrate the route invocation process as illustrated in Figure 11. Suppose the application at node N_p decides write a value v to data item d. The application invokes the `Write`(d, v) method. As a result, the unified storage protocol invokes the `route`(N_p, d) method that returns with the next node's identifier N_q. Then N_p forwards a write message to N_q. When this message arrives at the unified storage protocol of N_q, it invokes the

$\texttt{route}(N_q, d)$ that returns with N_r. The write message is then forwarded to N_r. The latter invokes the $\texttt{route}(N_r, d)$ method that returns with N_s. Finally, when N_s invokes the $\texttt{route}(N_s, d)$ method, it returns with N_s. This indicates to the unified storage mechanism that it is responsible for data item d, and it stores the value v for d in its local storage module: if there was a previous value, then v overwrites it; otherwise, N_s initiates a new copy of d with value v. Figure 12 illustrates the same calling chain, but when the DHT module is instantiated with the Chord protocol.

[D1] Stoica I., Morris R., Liben-Nowell D., Karger D., Kaashoek M.F., Dabek F. and Balakrishnan H., Chord: A Scalable peer to peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17-32, 2003.

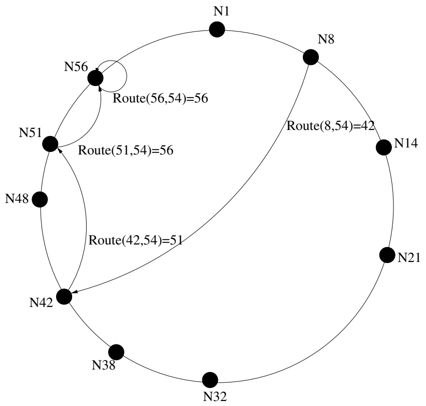


Figure 12: An example of `route` invocation process when instantiated for the Chord protocol (adapted from [D1]).