On the Benefits of the Functional Modular Approach to Distributed Data Management Systems (Position Paper)

Roy FRIEDMAN
Computer Science Department
Technion
Haifa 32000
Israel
`roy@cs.technion.ac.il`

Michel RAYNAL
IRISA
Université de Rennes 1
Campus de Beaulieu
35042 Rennes Cedex
France
`raynal@irisa.fr`

Abstract

Decomposing distributed data management 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 presents the first attempt at such a decomposition for peer-ro-peer systems.

Keywords: *Distributed Data Management, Modules, Distributed Shared Memory, Consensus, Peer-to-Peer*

1 Introduction

Designing, building, and reasoning about robust distributed data management systems are complex tasks. Yet, when considered closely, one can identify several different sources of complexity. Using the terminology of Brooks [6], some of these difficulties are *incidental* 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 abstractions. Simply put, much of the accidental complexity is a result of poor engineering approaches to these problems.

In this paper, we claim that applying good engineering approaches to representing, designing, and reasoning about distributed data management systems not only helps in their implementations, but also makes them more accessible for theoretical studies. Theoretical studies help understand the limitations of such systems, enables specifying precisely what they do, and consequently, formally proving their correctness.

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. In particular, 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 investi-

---

[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.
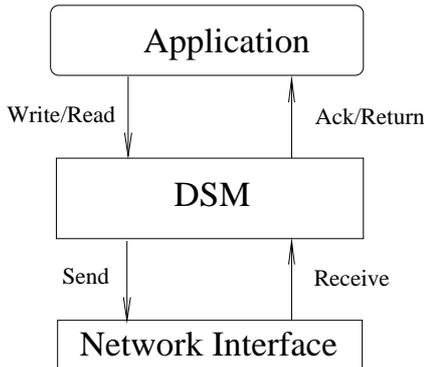
Figure 1. A Typical Modular View of DSM

gate this problem in a formal manner. For example, it is possible to investigate what are 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.

We demonstrate this thesis by examining two established domains, namely distributed shared memory and distributed agreement problems, including the Consensus problem. In both domains, we survey initial models and specifications that were given in a non-modular approach, and discuss the benefits of their functional modular alternatives. We then turn our attention to peer-to-peer systems. Despite being a new domain in distributed data management, existing works on peer-to-peer fail to use the functional modular approach. As a result, they suffer from the same deficiencies as early works on distributed shared memory and on agreement problems. Namely, the implementations are described in a monolithic manner and there are no clean formal models describing what these systems do. Consequently, it is hard to understand the precise environmental assumptions required for these systems to work, and there are typically no complete rigorous proofs of correctness.

2 Distributed Shared Memory

Initial works on Distributed Shared Memory (DSM) like Release Consistency [15], Alpha consistency [1], and Java Consistency [17] have mixed implementation details with specifications. This resulted in specifications that were extremely difficult to understand

and prove.[2] On the other hand, the approaches taken in works on Sequential Consistency and Linearizability [5], Hybrid Consistency [4], Causal Ordering [2], the non-operational definition of Java Consistency [16], and Normality [14], have proposed to make a clean separation between implementation details and functionality. In these approaches, the system is decomposed into several modules, which include a *DSM emulation module*, an *application layer*, and a *network interface module*, as illustrated in Figure 1.

The DSM module is equipped with a given interface, which includes methods that can be invoked by the application, 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 emulation module 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 [2, 4, 5, 12].

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. Moreover, it allowed 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.
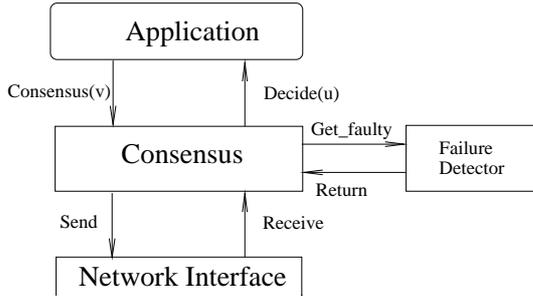
Figure 2. A Modular View of an Asynchronous System Enriched with a Failure Detector

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 [11]. 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 several *timed-asynchronous* [9] and *quasi-synchronous* models, with explicit timing assumptions. 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.

A cleaner modular solution was proposed by Chandra and Toueg, who introduced in their seminal work the notion of a *failure detector* [8]. 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. This mechanism enabled identifying precisely what are the minimal levels of completeness and accuracy that are required to solve various agreement problems [7, 10, 18].[3] Similarly, it 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, e.g., [8, 21, 25].

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, denoted $\diamondsuit\mathcal{S}$, which enables solving Consensus in otherwise asynchronous distributed settings, can be also implemented in timeless environments [20]. 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 our thesis, 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 $\diamondsuit\mathcal{S}$ work correctly in both environments, each with its corresponding implementation of a $\diamondsuit\mathcal{S}$ module. This again exhibits the benefits of the functional modular approach.

---

[2]In the case of Java consistency, as was shown in [23], several commercial compilers by major vendors did not implement correctly the memory model of Java according to its initially unclear specification.

[3]Friedman, Mostéfaoui, and Raynal have recently shown that by adding a second "dummy" method to this interface, it is possible to weaken these requirements even further [13].

4 Peer-to-Peer

Peer-to-peer is an emerging area of distributed computing. In peer-to-peer systems, the end users of the system communicate directly with each other, and provide the system's 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. Thus, much of the work on peer-to-peer involves designing schemes that enable the system to operate while each node only needs to know about a small fraction of the whole system. In particular, most implementations rely on Distributed Hash Tables (DHT) [22, 24, 26, 27].

Disappointingly, despite being a new domain, existing research on peer-to-peer suffers from the same poor engineering practices of initial works in other distributed computing domains. That is, there is no clear separation between levels of abstractions, no clear specifications of what these systems are doing, and in particular, no specification of the exact conditions under which the system will behave as promised.

In this paper we propose a potential functional modular decomposition of peer-to-peer systems. Our architecture, which extends [3], may not be the only one, but it serves as a proof of concept. That is, 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.

Unfortunately, for lack of space, the discussion below is not completely formal, and sometime even slightly inaccurate. The exact complete details appear in the full version of this paper. In general, many of the missing details can be seen as natural adaptation of works such as [5] to dynamic settings.

4.1 Problem Statement

We assume a distributed environment composed of a finite but unbounded and changing set of processes. Each time the set of processes changes, we call it a *configuration change*. The goal of the system is to implement a *semi-reliable distributed storage mechanism*. That is, we call a collection of sequences of read and write operations, each of these sequences executed by a single process, a *partial execution*. If during a partial execution there are no failures or configuration changes, we say that it is an *uninterrupted partial execution*. We then require that for each uninterrupted partial execution σ of the system, it is possible to find a total order ϕ that
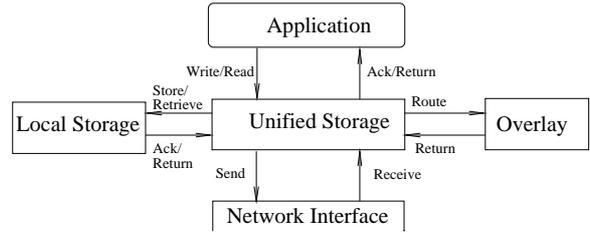

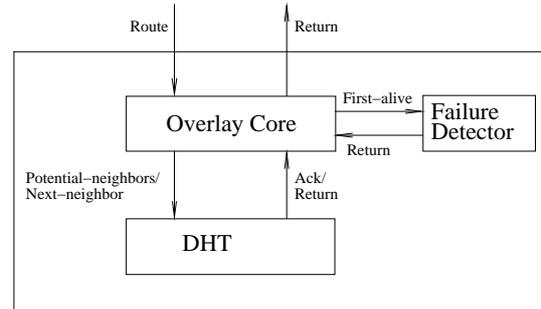**Figure 3. A Modular View of a Peer-to-Peer System**

Figure 4. Decomposing the Overlay

extends the real time ordering of all operations in σ such that each read operation returns the value written by the last write that precedes it in ϕ. This is similar to the notion of *linearizability* as defined in [5, 19], restricted to an uninterrupted partial execution.

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]

4.2 Architecture, Specifications, and Implementation

Figure 3 illustrates a modular view of peer-to-peer systems. As can be seen, here all nodes are symmetric. Each node is composed of four parts: the *application*, a *unified storage* module, an *overlay* module, a *local storage* module, and a *network interface*. The unified storage module plays a similar role with respect to the application and network module as the DSM module of Section 2.

---

[4]An example of a fault-tolerant version of such a storage mechanism under slightly different assumptions appears in [3].

4

```
Upon Write(o, v) from the application do
    ts := ts + 1;
    p_l := overlay.Route(p_i,o);
    send (WRITE,o,v,(ts, i)) to p_l;
    return
enddo

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 := max(ts,t);
    return v
enddo

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, 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, 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
enddo

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
enddo
```

Figure 5. A Generic Peer-to-Peer Protocol

However, the unified storage module can consult the overlay module on how to route read/query and write/update requests coming from the applications and the network. That is, the overlay module exposes a single method called route, which accepts an object identifier and the current process id and returns the next process id 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. 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 mod-

ule uses the local storage module. The exact protocol is summarized in Figure 5 (with ts initialized to 0 before the execution begins).

In order to formally define the overlay's properties, we first define the following concepts: A *targeted invocation sequence* is a sequence of invocations of the Route methods such that (a) all invocations are called with the same object identifier, and (b) the process id passed to the $(i + i)^{\text{th}}$ invocation is the one returned by the ith 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 overlay's Route method must satisfy the following properties:

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.[5]

Internally, the overlay module is based on three submodules: the *overlay core* module, the *DHT* module, and a failure detector module. The overlay core module is responsible for running the main protocol and is aided by the other two modules. The DHT 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 of the processes appearing in its input list. The failure detector component supports one method: First-alive; it accepts an ordered list of processes and returns the id of the first process on the list that is alive. The implementation of the overlay based on the DHT and failure detector modules appears in Figure 6.

Due to space limitations, we skip the definitions of the DHT module and the failure detector module spec-

---

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

```
Periodically do
    candidates := DHT.Potential-neighbors(p_i)
    empty finger
    foreach entry k in candidates do
        finger[k] := FD.First-alive(candidates.[k])
    enddo
enddo

Route(obj,p_i)
    return DHT.Next-neighbor(p_i,finger, obj)
```

Figure 6. A Generic Peer-to-Peer Protocol

ifications. The exact definitions appear in the full version of this paper. Yet, they can be viewed as extensions of [3] and [13]. Also, in mapping various peer-to-peer systems to our model, 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.

References

[1] *The Alpha Architecture Handbook*. Digital Equipment Corporation, 1992.

[2] M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1), 1993.

[3] E. Anceaume, R. Friedman, M. Gradinariu, and M. Roy. An Architecture for Dynamic Scalable Self-Managed Distributed Transactions. In *Proc. of the 6th IEEE International Symposium on Distributed Objects and Applications (DOA)*, October 2004.

[4] H. Attiya and R. Friedman. A Correctness Condition for High-Performance Multiprocessors. *SIAM Journal of Computing*, 27(2):1637–1670, April 1998.

[5] H. Attiya and J. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[6] F. P. Brooks. *The Mythical Man-Month*. Addison Wesley, October 1995. 20th Anniversary Edition.

[7] T. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[8] T. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. *Journal of the ACM*, 43(4):685–722, July 1996.

[9] F. Cristian and C. Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Transanctions on Parallel and Distributed Systems*, 10(6):642–657, June 1999.

[10] C. Delporte, H. Fauconnier, and R. Guerraoui. Failure Detection Lower Bounds on Registers and Consensus. In *Proc.*

[11] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[12] R. Friedman. Implementing Hybrid Consistency with High-Level Synchronization Operations. *Distributed Computing*, 9(2), 1995.

[13] R. Friedman, A. Mostefaoui, and M. Raynal. Asynchronous Bounded Lifetime Failure Detectors. Technical Report PI-1628, Insisute De Recherche En Informatique Et Systems Aleatoires (IRISA), Rennes, France, June 2004.

[14] V.K. Garg and M. Raynal. Normality: A Consistency Condition for Concurrent Objects. *Parallel Processing Letters*, 9(1):123–134, 1999.

[15] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[16] A. Gontmaker and A. Schuster. Non-Operational Characterizations for Java Memory Model. *ACM Transactions On Computer Systems (TOCS)*, 18(4):333–386, November 2000.

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

[18] R. Guerraoui and P. Kouznetsov. On the Weakest Failure Detector for Non-Blocking Atomic Commit. In *IFIP TCS*, pages 461–473, 2002.

[19] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[20] A. Mostefaoui, Eric Mourgaya, and Michel Raynal. Asynchronous Implementation of Failure Detectors. In *Proc. of the 4th IEEE Conference on Dependable Systems and Networks*, pages 351–360, 2003.

[21] A. Mostefaoui and M. Raynal. Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. In *Proc. of the 13 Symposium on Distributed Computing*, pages 49–63, 1999.

[22] C. Plaxton, R. Rajaram, and A. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

[23] W. Pugh. Fixing the Java Memory Model. In *Proceedings of the ACM Java Grande Conference*, June 1999.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large Scale Peer-to-Peer Systems. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[25] A. Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10:194–157, 1997.

[26] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM 2001*, August 2001.

[27] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical Report UCB/CSD-01-1141, Computer Science Department, U.C. Berkeley, April 2001.