MOMMIE Knows Best: Systematic Optimizations for Verifiable Distributed Algorithms

Petros Maniatis
Intel Labs Berkeley

Michael Dietz
Rice University

Charalampos Papamanthou
Brown University

1 Introduction

Complex distributed algorithms become running systems through an integration with optimizations that target the system's deployment environment. Although expedient, this approach has disadvantages. First, this often makes implementing the algorithm difficult, since its logic must be composed with the optimizations. Second, proving the guarantees of the implementation is tedious, because the proofs must be derived for the composed algorithm, which may not be directly mappable to the original, unoptimized algorithm. Finally, retargeting the implementation to a different deployment—requiring a different set of optimizations—can be wasteful, since a new composed algorithm must be derived to include this different set of optimizations, including their correctness proofs.

We fault the tussle between abstraction and performance as the fundamental cause for this problem. On one hand, algorithm designers need abstraction to simplify the job of identifying and proving invariants or liveness properties about their algorithms. On the other hand, algorithm implementers have faced an age-old conviction (grounded in at least some truth) that abstraction hurts performance, which for the complexity and latencies involved in distributed algorithms often leads to unusability.

This trend is evident in the level of detail present in the pseudo-code descriptions of many replicated systems. For example, the PBFT system [4], a replicated state machine based on Byzantine consensus, gives an I/O Automaton specification—arguably a clean formalism intended for proving properties—that nevertheless contains explicit details about cryptographic tools (digital signatures vs. MAC authenticators), content dissemination (push vs. pull, direct broadcast vs. one-hop flood), deduplication (send whole object vs. object digest), transport mechanisms (retransmissions, negative acknowledgments) and group communication (manual collection of identical messages from many sources to form quorums).

Beyond specification, these implementation details feature prominently in the correctness proofs prepared by the authors; in fact, the authors had to prepare distinct proofs for different versions of their algorithm that differed in the choice of implementation details (crypto for message authentication) but ostensibly not in the fundamental "business logic" of the algorithm. This was not an isolated example: in the decade since that work, dozens of other proposals have been presented that push into the business logic further optimizations for distinct execution environments and workloads (e.g., mostly non-faulty or trusted nodes, or even mostly read-only or contention-free write requests). Although such overspecialization is successful at improving performance, at an algorithmic level it increases complexity and hurts dependability: implementations appear with missing features, and correctness proofs end up lacking rigor.

In this paper we challenge the validity of the tussle at the heart of this trend. We argue for a different way to build replicated systems that are both rigorous in specification and guarantees, but also optimized to perform well in their particular execution environment and workload. We make the case for separating implementation details from the algorithms they optimize—arguably as old an idea as computer science. In particular, we advocate a replicated algorithm abstraction model that admits provably safe, plug-and-play, fine-grained optimizations that can be chosen and applied automatically, maintaining any safety and liveness guarantees proven for the optimization-free algorithm specification.

2 What To Optimize

Consider the following toy example, reminiscent of many primary-backup replicated systems (Figure 1). There is a number of server replicas and an elected (usually, rotated) primary. When the application requests a service call (Step 0), the client sends a request to all replicas (Step 1). The primary assigns a sequence number to a received request to order it globally, and notifies all replicas of the assigned order (Step 2). Replicas execute requests in the sequence-number order assigned by the primary (Step 3), and then send a reply to the client with the execution results. The client waits for a quorum of replies (say 2/3 of all replicas) with the same result (Step 4), and then notifies the application (Step 5). If a replica receives no ordering within a short time after it has received a client request, it suspects the primary of malfunction (Step 6).

This toy design is a simple enough algorithm, yet in practice it would have to be augmented in several respects to improve performance, e.g., client-observable response times, and to reduce overhead, e.g., bandwidth and CPU utilization. This augmentation, depending on deployment assumptions, results from different choices of how to im-

This paper appears in HotOS 2011.

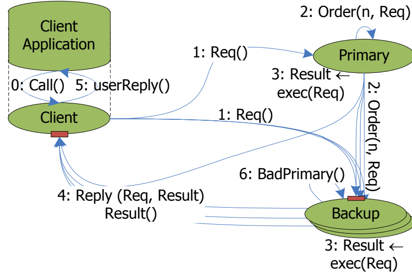

Figure 1: A simplified primary-backup system.

plement the business logic; we call these *implementation idioms*, and describe a few next.

First, the client might send requests only to the primary (in Step 1), which forwards them to replicas (sparing client-replica bandwidth), or the client might broadcast the request to all replicas, as shown, avoiding overload on the primary but using more bandwidth.

Second, the ordering message in Step 2 may bundle the whole request along for the ride, taking up bandwidth but ensuring both ordering and request will be available at the recipient replica; instead the ordering message may only contain a cryptographic digest of the request (for integrity), allowing the request to be demand-fetched by the backup, resulting in higher latency due to the extra step between Steps 2 and 3.

Third, authenticated messages that must be *non-repudiable*—i.e., verifiable by all—to ensure system liveness in Byzantine settings may be digitally signed with an expensive asymmetric cryptosystem such as RSA. Alternatively, they may achieve non-repudiation via cheaper point-to-point authentication (e.g., keyed hashes) but with additional assumptions (a threshold fault model) and protocol complexity (e.g., atomic broadcast).

Fourth, the implementation of send-to-all message semantics from the client to replicas in Step 1 may be implemented as broadcast, resulting in worst-case bandwidth utilization. Otherwise, the sender may send to only some of the replicas hoping it will be enough, and send to the rest if no progress is made—this appears in "preferred-quorum" optimizations in several systems.

Finally, the primary may transmit Order messages to backups one at a time, paying per-message cryptographic and network costs, or it may delay earlier messages to batch them together amortizing costs. Other such idioms have different performance implications [16].

Beyond implementation, there are some common *expressive idioms* that creep up almost always when building replicated systems. For example, often a replicated server must wait until it receives the same message from multiple other replicas before it can proceed to the next step of the algorithm (we call this *data rendez-vous*); and fall-back actions (e.g., leader election) occur when timeouts or other conditions arise. Such expressive idioms are required no matter how an algorithm is implemented, and usually result in extra specification complexity (e.g., extra transitions in a state-machine formalism) only to perform rather boiler-plate, algorithm-unspecific manipulation of messages. In other domains, entire classes of programming languages (event-coordination languages [7]) exist to capture precisely such rendez-vous semantics.

All of the idioms we have described have a *coordination* flavor: they are about messaging among processes, not about computation within a process. Furthermore, they are independent of the semantics of the algorithm in question: the designation of primaries, clients and backups is immaterial to the functioning of the idioms. Others have proposed deeper algorithmic idioms, that attempt to refactor and compose algorithms with each other, as opposed to optimizations to algorithms [2, 8]. We believe those to be just as important, but limit ourselves to coordination idioms, which are simpler to abstract, combine, and systematically manipulate aggressively.

3 Design Considerations

Embarking into this project, we considered the following goals essential to success. First, *provable guarantees*, not only for the algorithm but for as much of the running system as possible, are fundamental to achieving dependability. Second, any solution should enable *retargetable* systems: change in the deployment characteristics should not require building up an algorithm, correctness proofs, and implementation from scratch. Third, perhaps unconventionally, we favor *skill separation*, in which deployment engineers need not necessarily understand deeply the algorithms they are trying to optimize for their environment; conversely, neither should algorithm designers be well versed in the tricks of building efficient implementations. Finally, we are targeting a *smooth adoption curve* for algorithm designers and system builders, alike.

Closest to our goals comes MACE [9]. It specifies distributed systems in a macro language, which is then compiled into C++ (or other languages). MACE programs can be model-checked for debugging both safety and liveness violations. Although useful to many researchers over the years, MACE does not meet all of our goals. It offers no provable guarantees, but uses model-checking to find bugs; absence of bugs does not mean correctness. It has not explicitly addressed retargetability. As a consequence, it does not address skill separation either. These shortcomings may be moot if the goal is to build an efficient, debuggable distributed system fast, but our goals are different.

We considered several design trade-offs, mostly exemplified by systems and techniques used in the past.

Programming Discipline—Declarative networking (e.g., systems like P2 and others [12, 14, 16, 17]) has been proposed for routing protocols, overlays, distributed systems, and even cluster services. Its claim to fame is using

a high-level declarative language (e.g., logic or functional languages) to program systems. On the other hand, declarativity is relative [15], and often alienates programmers: relational elegance aside, not everything is a relation.

In our approach we chose a hybrid design, in which we allow programmers to retain an imperative programming style for computation—the parts of an algorithm typically executed locally within a single node—but use a very high-level declarative abstraction for coordination. The rationale is that coordination among processes in complex replicated systems is increasingly effected via RPC libraries, and group communication primitives. Such tools typically auto-generate code from specification. Programmers tend to be less "possessive" of the minute details in coordination. This is consistent with our goal of a smooth adoption/learning curve.

System View—A programming abstraction that offers programmers a single-system illusion may significantly simplify algorithm specification. In a traditional network-of-computers paradigm, a client may have to prepare a request and transmit it to a server where the server validates the request before executing it and responding. In contrast, a shared-everything global-view paradigm can abstract all this as a single function call or variable assignment. On the other hand, maintaining the single-system illusion implies complex, subtle understanding of the semantics of the algorithm with respect to state characteristics, access patterns to different objects, consistency expected, etc.; a shared-nothing paradigm may instead better capture isolation among logically distinct components.

We chose to adopt a local-view system model in which nodes only implicitly share immutable coordination data; shared updatable objects, if required, must be built on top of the basic programming paradigm. This is in stark contrast to other high-level distributed programming paradigms [13, 14], but is most consistent with how replicated algorithms are designed and analyzed today—typically in some form of event-driven state-machine representation. It also simplifies the semantics required for coordination among nodes: if nothing mutable is shared, the programming abstraction need not encompass complex consistency models, and enables sophisticated caching to improve performance.

Level of Abstraction—Abstractions come at different levels of detail, enabling the composition of business logic and optimizations of differing scope and sophistication. For example, a primitive for message authentication could rest its abstraction at the cryptosystem level (i.e., "RSA-signature") abstracting implementation away; this is how "providers" work in cryptography toolkits such as Java's JCE. Alternatively, the primitive might lie at the mechanism level (i.e., "public-key signature") abstracting the choice of cryptosystem (as in PBFT-like protocols in the literature), or even at the authentication property level (i.e., "non-repudiable authentication"), which may be implementable with cryptographic or other mechanisms (e.g., Matrix Signatures [1]). Intuitively, the choice of abstraction level should match the common expressive building blocks that algorithm designers use. For replicated algorithms, the choice of cryptographic algorithm is almost always irrelevant.

Our choice was to push the level of abstraction up aggressively over that used in most replicated systems in the literature. We opted for authenticated multi-sender-multi-receiver messages with references. We introduced a generalized authentication primitive that abstracts implementation mechanisms away from the algorithm specification; this allows the seamless composition of both cryptography- and protocol-based tools to implement authentication. Our messaging primitive abstracts details of multiple senders (used in quorum statements in replicated protocols), as well as the more traditional notion of multiple receivers (which can be mapped to network multicast or unicast, as well as sophisticated content dissemination techniques). Finally, we abstract away the means for implementing message-to-message references (typically implemented either via digests or via explicit nesting).

4 MOMMIE Knows Best

Guided by our goals and design choices, we are currently exploring a systematic yet practical decomposition of the logic of distributed algorithms from optimizations, providing the following benefits. First, algorithm designers need only design the inherent algorithmic logic—*computation* within computing nodes and *coordination* among nodes—in a single, high-level programming framework logically reminiscent of tuple spaces [7], which we call *MOMMIE* (for MOMMIE Optimized Messaging Middleware in Insecure Environments). A MOMMIE program can be used for verification and reasoning about algorithm properties, since it can be translated directly to a formal specification (we use TLA+ [11]). In TLA+ properties can be rigorously proven by hand, using theorem provers [6], or model-checking [11, Ch.14]. The same MOMMIE program can be used to generate executable code that runs within a deployment platform (computers and networks). In fact, a programmer may choose to use a more traditional way to code up specific parts of the computation (e.g., in C++), and use the MOMMIE program as a way to generate runtime assertions for those hand-coded fragments.

Second, a deployment engineer can automatically compose the MOMMIE program with her desired pre-existing or custom *implementation modules* for optimized coordination, such as those described in Section 2. The composed, optimized runtime preserves the safety and liveness properties proven for the algorithm; the engineer

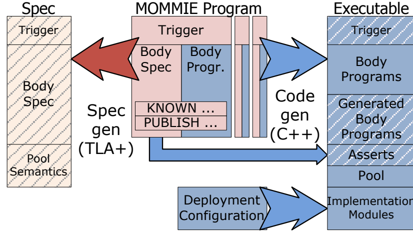

Figure 2: A MOMMIE program (center), can be translated to TLA+ (left) or executable code (left, along with configured implementation modules). Diagonal shading indicates auto-generated components.

need only be concerned with picking the modules that improve performance for a particular deployment environment. In practice, the choice of implementation modules may also be done via mathematical or heuristic optimization. Figure 2 presents a high-level illustration of how a MOMMIE program can become a formal specification or an optimized executable.

Programming Abstraction—A MOMMIE program consists of a number of state transitions, like the following example transition for a replica's Step 3 from Figure 1.

```
1 WHEN [PRIMARY, ME, NONE] Order order {{ m==req; seqNo==executeSeqNo; }}
2 ANDWHEN [c C == 1, ME NOONE] Request req
3 @@{
4    Result res <- execute(req);
5    PUBLISH [ME, c, NOONE] Reply(req, res);
6    executeSeqNo <- executeSeqNo + 1;
7 @@}
```

The transition consists of a trigger describing the enabling condition for the transition (`WHEN ... ANDWHEN ...`), and the body spec (`@@{ ... @@}`) executed when the transition is actually invoked. The body spec appears like an imperative sequence of simple commands, including variable assignments, conditionals, set comprehensions, etc., as well as `PUBLISH` commands that emit information for other nodes, as described below. A designer may also provide a *body program* in a language such as C++, in addition to the body spec; while the body spec will be used for verifying the properties of the algorithm, the body program will be used in the executable, with interwoven runtime assertions derived from the body spec. In the absence of a body program, executable code for the body of a transition is auto-generated from the body spec.

Information is conveyed among nodes in *authenticated statements* of the form `[I, V, A]X`, where `I`, `V`, and `A` are the statement's issuer, intended verifier, and intended auditor (principals), and `X` is a record similar to a C struct. Statements move from node to node via the MOMMIE *pool*, a logical collection of interconnected per-node statement caches. A node `i` `PUBLISH`es a statement `[i, V, A]X` into the pool, and that statement is eventually made `KNOWN` to its intended verifier `V`. Auditors are meant to capture the security notion of *non-repudiation*. Any statement `[I,V,A]X` can be verified not only by its intended verifier `V` but also by its auditor `A`; unlike its verifiers, though, auditors are not intended as immediate destinations for such statements. Line 6 in the example above emits a statement of type `Reply` authenticated by the local node (system constant `ME`), and intended to be verified by node `c` (formal parameter of the transition declared in line 3), and auditable by no one (system constant `NOONE`).

A node may inspect its local pool synchronously, via the `KNOWN` operator, or asynchronously via transition triggers in `WHEN` expressions. In both cases, the lookup is specified as a *statement pattern*, a statement with some of its authentication principals or fields unbound. We will forego details due to space constraints, but as an example, line 1 above expects statements whose issuer matches the constant `PRIMARY` (declared elsewhere), its verifier is the local node, it may or may not be repudiable (no auditor is expected), and matches the record type `Order`, whose fields `m` and `seqNo` are bound by equality constraints; other field constraints not shown include univalent and multivalent don't-cares. A distinctive feature of lines 1 and 2 (a conjunction of statement patterns) is that the field `m` of the expected `Order` must match the `Request` record expected in line 2; this is a message-to-message reference. In practice, MOMMIE may implement this reference via a cryptographic digest, or via nesting of the Request within the authenticated Order message. Another notable type of statement pattern concisely abstracts quorum messages, as in the client's Reply collection (Step 5).

```
1 WHEN [R >= 3, ME, NONE] Reply r
```

The client expects the same `Reply` record issued by at least 3 of the principals from the set of all replicas `R` (a constant declared elsewhere). The programmer need not explicitly write out state transitions that collect one reply after another, checking that eventually 3 are collected; the middleware takes care of that. The MOMMIE pool logically derives authenticated statements according to a number of statement inference rules reminiscent of BAN logic [3]. For example, issuers from `[I1,V,A]X` and `[I2,V,A]X` in a pool are merged automatically to derive `[{I1,I2},V,A]X` in that pool.

Optimization Interface—A deployment engineer (or a mathematical optimizer) can specify a deployment configuration of the form

```
1 [Authentication]                     5 y -> *          : RSA
2 a <-> b        : Plain, HMAC-SHA1    6 [Dissemination]
3 [Non-repudiation]                    7 x,y -> *        : OPTIMISTIC(PRIMARY);
4 x -> *         : TTP(z)              8                   OPTIMISTIC(QUORUM);
```

that determines what mechanism, cryptographic or otherwise, to use for authentication and non-repudiation between two principals, and how to decide where to forward authenticated statements in practice. For each authentication/non-repudiation mechanism, we establish what kind of authenticated statements it provides once, which we subsequently check at deployment time. For example, HMACs provide authenticated but repudiable (auditor is `NOONE`) statements, whereas RSA signatures provide universally auditable statements. We also consider

plaintext authentication (when the deployment engineer has determined an authenticated channel, such as a VPN connection, already exists), trusted third parties for non-repudiation, etc. Notably, each principal-to-principal pair may be configured with a distinct authentication mechanism by the deployment engineer, yet the high-level protocol will be seamlessly able to operate using abstract authenticated statements, quorums thereof, etc. The safety properties of the abstract algorithm are maintained due to the composition with authenticated statement implementations that individually retain authentication semantics.

A data dissemination plan is chosen by turning authenticated statements into distinct network messages. Dissemination is configured as a sequence of dissemination optimizers. Each optimizer is tried in order with a configurable escalation timeout. If the optimizer succeeds in getting the publisher's state machine to make progress, no other optimizers are used; otherwise, MOMMIE chooses the next optimizer in the sequence. When all optimizers have been exhausted, MOMMIE eventually reverts to the most general dissemination mechanism, unicast to all intended statement verifiers. Since this fallback implements precisely the semantics of published statements, MOMMIE preserves the liveness guarantees of the algorithm; dissemination has no impact on safety.

An interesting feature of MOMMIE is that `PUBLISH` may be accompanied by a statement pattern whose satisfaction indicates the achievement of "progress" (e.g., a transition into a new state); this allows the dissemination mechanism underneath to know when to stop trying further network exchanges for the same abstract authenticated statement.

5 Outlook

MOMMIE is at the early stages of development. It is more suspicion rather than a proof of feasibility. We have studied its theoretical underpinnings including its semantics and the properties of the abstract and optimized pool. We have created an initial prototype that parses the language, generates TLA+ specifications, and executable optimized runtimes (for very simple body specs). Much remains to be done to evaluate the price of abstraction, and the benefits of systematic optimizations.

More specifically, although we have shown how an algorithm can turn into a running protocol, we have not yet attempted to develop a full software stack. Until then, any properties "guaranteed" by MOMMIE are at best approximations of a world in which kernels, drivers, libraries, and applications act according to spec. The progress made by seL4 [10] encourages and inspires us, but the massive efforts involved point to a long journey ahead.

With this work we produce no new algorithms or protocols. Instead, our vision is to bridge the gap between those who think hard to develop and prove new distributed algorithms, and those who work hard to implement such algorithms into efficient, performant systems on complex environments. In a sense, we hope to address a complaint raised by Chandra et al. at the 2007 PODC symposium: "The fault-tolerance computing community has not developed the tools to make it easy to implement their algorithms" [5]. Our secondary goal is to help harness the optimization tricks devised for one distributed algorithm and apply them easily to another; it is telling that in the same work quoted above, Chandra et al. reinvent numerous optimizations described and implemented a decade before. Our final, perhaps less noble goal is to liberate distributed systems researchers—and corresponding academic publication venues—from the seemingly unquenchable thirst to re-specify, re-validate, and re-implement the same classic distributed algorithms in yet another slightly different deployment setting with slightly different optimizations, allowing them to focus on algorithmic innovation instead.

References

[1] A. S. Aiyer, L. Alvisi, R. A. Bazzi, and A. Clement. Matrix Signatures: From MACs to Digital Signatures in Distributed Systems. In *DISC*, 2008.

[2] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I Do Declare: Consensus in a Logic Language. In *NetDB*, 2009.

[3] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM TOCS*, 8(1), 1990.

[4] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM TOCS*, 20(4), 2002.

[5] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *ACM PODC*, 2007.

[6] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying Safety Properties With the TLA+ Proof System. In *IJCAR*, 2010.

[7] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Commun. ACM*, 35(2), 1992.

[8] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The Next 700 BFT protocols. In *EuroSys*, 2010.

[9] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[10] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP*, 2009.

[11] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[12] L. Leonini, É. Rivière, and P. Felber. SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze). In *NSDI*, 2009.

[13] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A Platform for Secure Distributed Computation and Storage. In *SOSP*, 2009.

[14] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[15] Y. Mao. On the Declarativity of Declarative Networking. In *NetDB*, 2009.

[16] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT Protocols Under Fire. In *NSDI*, 2008.

[17] W. Zhou, Y. Mao, B. T. Loo, and M. Abadi. Unified Declarative Platform for Secure Networked Information Systems. *ICDE*, 2009.