Simplifying Distributed System Development

We predict that the future is in . . . predicting the future.

Maysam Yabandeh, Nedeljko Vasić, Dejan Kostić and Viktor Kuncak
School of Computer and Communication Sciences, EPFL, Switzerland
email: `firstname.lastname@epfl.ch`

Abstract

Distributed systems are difficult to design and develop. The difficulties arise both in basic safety correctness properties, as well as in achieving high performance. As a result of this complexity, the implementation of a distributed system often contains the basic algorithm coupled with an embedded strategy for making choices, such as the choice of a node to interact with.

This paper proposes a programming model for distributed systems where 1) the application explicitly exposes the choices (decisions) that it needs to make as well as the objectives that it needs to maximize; 2) the application and the runtime system cooperate to maintain a predictive model of the distributed system and its environment; and 3) the runtime uses the predictive model to resolve the choices so as to maximize the objectives. We claim that this programming model results in simpler source code and lower development effort, and that it can lead to increased performance and robustness to various deployment settings. Our initial results of applying this model to a sample application are encouraging.

1 Introduction

The difficulties in developing distributed systems arise both in basic safety correctness properties (as exemplified by the design of the Paxos algorithm [6]), and in achieving high performance across a wide range of deployment settings [11]. Developers are likely to increasingly face these problems as the vision of cloud computing starts connecting data and computation across a set of data centers that are spread over the wide area network.

The implementation of a distributed system often contains the basic algorithm coupled with a strategy for making *choices*. Examples of choice include choosing a node to join the system, choosing the node to forward a message to, or choosing how to adapt to a change in the underlying network. In current frameworks, such choices are buried in the application source code together with basic functionality. There is evidence [8, 7, 14] that changing the strategy for making such choices can result in better performance and resilience properties. To make informed choices, many applications perform measurements and derive a model of the distributed system [2, 5]. The importance of such measurements is supported by iPlane [10], which proposes to build an information plane that makes network measurements and predictions available to all applications on a given node.

We have recently proposed an approach [15] for collecting information based not only on the past behavior, but also by exploring possible future executions, then using this information to detect and avoid bugs. Because of a focus on finding bugs and increasing the resilience of *existing* systems, the impact of this approach was limited by the way in which the original service was written. This experience suggested an approach where the developer is aware of sophisticated runtime mechanisms, and uses them productively to reduce the development effort.

The hypothesis of this paper is that we can lower the development effort, and ultimately increase performance and robustness to various deployment settings by a programming model that 1) exposes the choices that the application needs to make, as well as the desired properties it needs to achieve 2) introduces the notion of a predictive system model that helps make appropriate choices.

We propose two main directions of future research to realize this approach:

1. Developing a programming model where the application exposes the choices it needs to make, and where the application and the runtime collaborate to create the model and use it to make predictions;

2. Developing new prediction techniques that integrate information from source code, past executions, and future executions, while leveraging increases in computational power and bandwidth. Such techniques need to be extended to at the same time take into account correctness in terms of specified or inferred safety and liveness properties, as well as performance.

As an initial example supporting our view, we describe the result of applying the proposed model to a sample application. We started from an existing, publicly released implementation of a random overlay tree, and replaced the hard-coded policies with exposed choices resolved by our generic runtime mechanism. Thanks to the predictions of our runtime mechanism, result was a significantly simpler implementation that performs as well as, or better than, the original version.

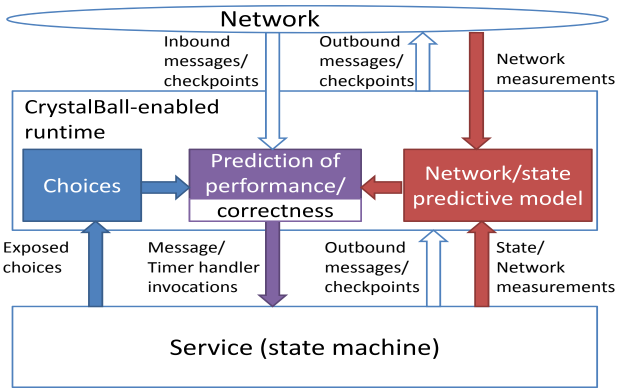

Figure 1: Overview of CrystalBall and its extensions. Existing components are shown as clear, extensions are shaded.

2 Background: CrystalBall

In this section, we briefly describe our existing work on predicting and preventing inconsistencies in deployed distributed systems using CrystalBall [15]. Figure 1 shows the high-level overview of a CrystalBall node (existing features are shown non-shaded). As in many existing approaches [4, 6], we assume that the distributed service is implemented as a state machine that runs on every node. The CrystalBall runtime interposes between the network on one side, and the existing runtime (that manages the timers and invokes the state machine handlers) and the state machine on the other.

CrystalBall has the ability to predict the reliability of the system. To accomplish this, the CrystalBall controller periodically collects a consistent set of checkpoints from each of the node's neighbors. The size of the neighborhood is typically limited in scalable systems, e.g., $O(\log n)$, but CrystalBall also works with systems with full global knowledge. The controller instantiates local versions of the participants' state machines from the checkpoints, and runs a state space exploration algorithm (termed *consequence prediction*) to predict which of the nodes' actions lead to violations of user- or programmer-specified safety properties. Consequence prediction focuses on exploring causally related chains of events, and is fast enough to look several levels of state space into the future fairly quickly (e.g., in 10 seconds) on today's hardware.

If the state-space exploration predicts an inconsistency, for example due to a message that changes the local state in a way that causes an inconsistency relative to the other participants' state, CrystalBall checks whether it is safe to *steer execution* away from the possible inconsistency. If consequence prediction does not find any new inconsistencies due to execution steering, the controller installs an *event filter* into the runtime. In case of messages, the event filter works by dropping the offending message and breaking the connection with the message sender for systems using TCP.

In summary, CrystalBall predicts actions that may cause inconsistencies in the near future, and prevents these actions by choosing certain universally possible alternatives, such as TCP connection failure. Our experience shows that the CrystalBall approach [15] effectively masks errors in deployed distributed systems. In doing so, this approach makes an implicit choice between standard actions and corrective actions. In this paper, we argue that such choices should become explicit in the programming model.

3 Towards an Explicit-Choice Architecture

Next, we describe the features of the new architecture (shown shaded in Figure 1).

3.1 Exposing Choices

Distributed systems often rely on a choice as a means of adapting to improve reliability and performance. We illustrate this through a number of examples.

Gossip Protocols. Nodes in epidemic dissemination protocols periodically pick a node from their views to exchange data. A random choice can improve reliability and help improve performance under some conditions. The work on BAR Gossip [8] shows that carefully restricting this random peer choice (to one node per round) can have a positive impact on the system performance in the presence of rational and Byzantine nodes. However, the performance might suffer if, e.g., the only target is behind a slow network connection. This system is an example of designing primarily for reliability. The follow-on to this work by the same authors [7] shows that it is necessary to relax the choice the nodes have in peer selection for better performance.

Overlay Trees. Given the lack of global (or even neighborhood) information, the programmer is often forced to use complex logic and to make random choices. In a random overlay tree (RandTree [4]), a node has the choice of forwarding an incoming join request to its parent or to one of its children, to meet the expected goal of a balanced tree. In this state machine-based implementation, a single message handler serves the join request. The logic for making the forwarding decision is fairly complex, and involves a few calls to a pseudo-random number generator.

Content Distribution. It is difficult to devise adaptive mechanisms that work optimally across a range of deployment settings, especially those in which the system has not been tested prior to deployment. The BulletPrime [5] and BitTorrent [2] content distribution systems have two different mechanisms for choosing the next block to request from any given peer, namely random and rarest-random. Experimental results show [5] that neither of these strategies is decidedly superior. BulletPrime chooses to run the rarest-random strategy, while BitTorrent has an ad-hoc mechanism to make a one-time switch from one to the other. This is an example of a case in which the system developer cannot predict all possible system environments and is forced to make some decisions. In other cases there might not be enough time to program complex adaptive mechanisms that will perform well. We argue that the distributed service should cooperate with the runtime, because the latter might have more information available, and is capable of predicting the performance and reliability.

In addition, systems often rely on random decisions to help them probe the environment. For example, the BitTorrent [2] nodes connect to a random subset of the existing participants to discover which file blocks are available. Potential peers are chosen via an external interface, i.e., a remote tracker which chooses a subset of peers from all the participants. Given this design choice, it was fairly straightforward to manipulate the peer choice made by the tracker [14] to bias it in a way that reduces ISP costs. Here, exposing the choice made it easy to improve system performance and meet ISP goals.

Consensus. The Paxos algorithm solves the consensus problem in the face of node and network failures. Its description [6] uses sufficient safety properties to implicitly present a non-deterministic algorithm, with many deterministic implementations possible. The original algorithm was successfully implemented and deployed in cluster settings. However, this algorithm does not offer a choice as to which node is allowed to propose a new value, and can suffer from reduced performance due to CPU overload or network congestion. A recent improvement [11] achieves significant performance gains across wide-area networks by allowing every node to propose according to a round-robin schedule. We argue that an implementation can expose the choice of a proposer and let the runtime pick the best proposer for high-performance across a range of deployment settings.

The New Programming Model. Based on all these examples, we claim that many distributed Algorithms already contain choices. However, current distributed system frameworks force the developer to resolve these choices, often leading to suboptimal decisions or complex code. We propose a programming model for distributed systems in which the programmer exposes the key choices to the runtime. For example, the programmer would expose the peer selection. The runtime can then consider several peers and return one. Another way of presenting the choices is to implement a distributed system as a non-deterministic finite state automaton (NFA) with multiple applicable handlers. Instead of hard coding the logic for making several choices into one message handler, the programmer can write several, simpler handlers for the same type of message. Each of the handlers is likely to be shorter as well as easier to maintain and reason about. It is then the runtime's task to resolve the non-determinism arising from multiple applicable handlers in a way that leads to good reliability and performance.

3.2 Exposing Objectives

To influence the strategy for resolving choices, the developer may specify the objectives that the runtime needs to maximize. Systems such as MaceMC [4] and CrystalBall already contain the ability to specify safety and liveness properties. Given such properties, a generically useful objective can be computed from the number of safety and liveness properties that are expected to hold at various points in the future.

Specifying performance objectives in a developer-friendly and flexible way is an important design question. The problem is especially interesting in the light of the weaker consistency guarantees provided by scalable distributed services, compared to their centralized counterparts. Such weaker properties are often best expressed in terms of performance. An expressive performance specification language can, in fact, subsume safety and liveness specification languages.

3.3 Predictive System Models

Central to our architecture is the notion of a system model, which incorporates information about system-wide state and contains performance parameters. Such a system model needs to be up-to-date, and needs to support mechanisms for efficiently resolving the choices to maximize the objectives.

3.3.1 Rationale for Exposing Models

To achieve good performance, distributed systems typically collect some information about the network and, often implicitly, build a *network model* to predict network performance. This is done either by explicitly probing various network conditions such as bandwidth, latency, and packet loss rate, or by passively inferring them. Every node also maintains some amount of local state, and collects information about other participants. We refer to this information as the *state model*. For example, a BulletPrime [5] node keeps track of the bandwidth to/from each of its peers, and the round-trip time file blocks take to be delivered from each sender. In addition to the network model, each receiver constructs a file map describing the set of available blocks at each of its sender. The applications typically feed the network and the state model to one or more adaptive mechanisms to predict performance and make an educated choice of action.

Keeping the collected information only within the application limits the runtime's ability to predict the future system behavior. We therefore argue that the network and the system model should be exported and kept in the runtime. This approach has the additional advantage of allowing the runtime to leverage other information services, enabling cost and overhead reductions when building a network performance model. For example, iPlane [10] proposes to build an information plane which makes the network measurements and predictions available to all applications on a given node.

Our long-term goal is to have developers implement new systems in the proposed programming model, and contribute to the models and parts of the runtime. Ultimately, we expect these to be reused across different systems, much like software libraries. While the added dependence on CrystalBall mandates that it is itself reliable, the added scrutiny over this shared component will help it achieve a reliability higher than that of protocol-specific individual system components, which are currently used by developers to resolve choices. Current frameworks result in time-consuming and expensive duplication of developer effort for building network and state models. In addition, building sophisticated models (e.g., to predict network performance) is difficult. We argue that it is better to make one concentrated effort to accomplish this task and let future developers leverage this information. We expect that such an effort will make the development of future systems easier, enabling developers with less expertise to build systems. They can also result in systems that adapt to various deployment settings, including those that have not been envisioned at development time.

Deciding on the data that should be kept in the runtime is potentially one new challenge that the programmer will face. Some of these decisions resemble those encountered when separating policy and mechanism in system design. Static and dynamic analysis tools might help identify parts of state that belong to the runtime.

3.3.2 Efficiently Maintaining Models

Our recent work revealed a number of challenges and possible solutions for maintaining useful system models at runtime.

How to deal with the lack of global information? Obtaining global information about the distributed system is a fundamental problem in distributed algorithms. To move the horizon beyond the currently collected *node neighborhood*, we propose the notion of a generic (*dummy*) node. The state of such a node is under-specified, which allows the model to explicitly take the partial nature of the available information. Taking into account the actions of generic node in principle requires the use of symbolic execution. Distributed service itself can contribute to efficiently maintaining the model by exporting state whose goal is to keep track of information in other nodes (e.g., file bitmaps in BulletPrime), and specifying how such a state could be computed. The runtime can then automatically maintain such information and use it to obtain a more accurate global model.

How to keep the model up to date? As the distributed system evolves, the model can become out-of-date. Moreover, the acceptable amount of communication overhead limits the rate at which information can be exchanged, especially in systems with rapid state changes. To overcome the fact that the model is not always up-to-date, our system starts from the latest known consistent snapshot, but also explores the near future of the system. To obtain information about the future, it currently uses an explicit-state model checker. To quantify the quality of the information in the model, it may be productive to incorporate confidence in the information as a function of its age.

How to model performance in detail? Previous results with distributed service-specific measurements and tuning suggest that there is a number of generic aggregate properties of interest that should be part of the model. This includes modelling the network, including latency, bandwidth, and loss information for the individual connections. Integrating this information into a state-space exploration algorithm turns a model checker into a simulator that runs a large number of simulations.

3.4 Using Models to Resolve Choices

Given a system model, the problem remains of how to use this information to maximize the objectives. Because objectives ultimately refer to future behavior (requiring that, for example, a state invariant will not be violated in the future), we use the model along with the system specification to predict the future behavior of the system in the environment. The challenge is that the predictions must happen fast enough to enable resolution of choices, without substantially slowing down the system. A useful design decision is removing complex mechanisms for making the choices from the critical path, using choices based on previous similar scenarios as a fast alternative, and updating the choices as more information becomes available. This design leverages the increases in computational power on multi-core machines.

We expect that new algorithms for online prediction of future behaviors will become available. Currently, we have used state space exploration up to a certain depth. Alternatives include variants of abstract interpretation adapted to deal with large initial states, as well as symbolic execution approaches. A useful way to speed up all these analyses is to precompute the impact of actions on system behaviors before the system is deployed. Such off-line computations can be performed using any of the currently existing approaches for static analysis.

Even with perfect prediction of future system states, optimizing complex performance-related objectives can pose non-trivial challenges, and may require the online deployment of constraint solvers that can deal with quantitative constraints [3]. Another challenge is the design of the execution steering module that avoids unwanted interaction and coupling among the system participants (e.g., emergent behavior [12]).

4 A Case Study

To highlight the benefits of exposing choice and using performance prediction at runtime, we replace the hard-coded policies with exposed choices in the released protocol for constructing a random overlay tree (RandTree), and contrast it with the baseline. Both systems are implemented in the Mace [4] framework. Our performance and correctness prediction is implemented within the Mace model checker, and we run the model checker as a separate thread. We conducted our live experiments with 31 participant over an Internet-like network using ModelNet on a cluster of dual-core machines.

Exposing choices results in a 43% decrease in lines of code (from 487 to 280). Using the number of if-else statements per handler to capture complexity, we observe that the complexity of the new code is 0.28, which is significantly lower than the baseline (1.94).

Next, we show that using the CrystalBall execution steering module to make the choices results in comparable or better performance. We install the objective that prioritizes building a balanced tree. We use two ways of resolving the choices in the new code: Choice-Random, and Choice-CrystalBall, which gives us a total of three different setups (including the Baseline). One of the metrics we use to observe the tree balance is maximum tree depth. After all 31 participants join the tree, the maximum depth is 6 in all cases (close to the optimal of 5). We then fail an entire subtree (about half of the nodes), and then let these nodes rejoin. Baseline and Choice-Random exhibit identical maximum depth (10), while the Choice-CrystalBall version is better with 9 levels.

5 Related Work

Win *et al.* [13] use simulated execution to verify distributed algorithms specified as nondeterministic I/O automata. Relative to these efforts, our work offers a mechanism for resolving nondeterminism at runtime. The general problem of optimizing objectives arises in the field of planning, which has also been related to the model checking [1].

There has been a large body of work on performance modeling, and our work will directly benefit from it. There has also been a substantial amount of work on performance tuning, to change system parameters based on past behavior. In contrast, our work predicts distributed system behavior at runtime, while taking into account the current state.

There has been recent work on program steering [9] by using machine learning techniques to let a program deal with unexpected inputs by matching an environment to program's mode of operation. This work does not consider distributed systems, and it does not try to anticipate future behavior.

6 Conclusions

We are calling for a programming model where the choices that a programmer of a given competence cannot easily resolve are left to a sophisticated runtime system. Such a runtime system accumulates techniques and algorithms that proved useful in a number of scenarios, and makes the developers of distributed systems more productive.

References

[1] M. R. A. Cimatti and P. Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159:127–206, 2004.

[2] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on the Economics of Peer-to-Peer Systems*, 2003.

[3] K. Keeton, T. Kelly, A. Merchant, C. Santos, J. Wiener, X. Zhu, and D. Beyer. Don't settle for less than the best: use optimization to make decisions. In *HOTOS*, 2007.

[4] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI*, 2007.

[5] D. Kostić, A. C. Snoeren, A. Vahdat, R. Braud, C. Killian, J. W. Anderson, J. Albrecht, A. Rodriguez, and E. Vandekieft. High-bandwidth Data Dissemination for Large-scale Distributed Systems. *ACM TOCS*, February 2008.

[6] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(121):18–25, 2001.

[7] H. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robinson, L. Alvisi, and M. Dahlin. FlightPath: Obedience vs Choice in Cooperative Services. In *OSDI*, 2008.

[8] H. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR Gossip. In *OSDI*, 2006.

[9] L. Lin and M. D. Ernst. Improving adaptability via program steering. In *ISSTA*, Boston, MA, USA, July 12–14, 2004.

[10] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: an Information Plane for Distributed Services. In *OSDI*, 2006.

[11] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *OSDI*, 2008.

[12] J. C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. In *EuroSys*, 2006.

[13] T. N. Win, M. D. Ernst, S. J. Garland, D. Kirli;, and N. A. Lynch. Using simulated execution in verifying distributed algorithms. *Int. J. Softw. Tools Technol. Transf.*, 6(1):67–76, 2004.

[14] H. Xie, Y. R. Yang, A. Krishnamurthy, Y. G. Liu, and A. Silberschatz. P4P: Provider Portal for Applications. *SIGCOMM Comput. Commun. Rev.*, 38(4):351–362, 2008.

[15] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.