WiDS: an Integrated Toolkit for Distributed System Development

Shiding Lin, Aimin Pan and Zheng Zhang
Microsoft Research Asia
{t-slin, t-aiminp, zzhang}@microsoft.com
Rui Guo†
Beijing University of Aeronautics and Astronautics
guorui@sei.buaa.edu.cn
Zhenyu Guo†
Tsinghua University
guozy03@mails.tsinghua.edu.cn

Abstract

Faced with a proliferation of distributed systems in research and production groups, we have devised the WiDS ecosystem of technologies to optimize the development and testing process for such systems. WiDS optimizes the process of developing an algorithm, testing its correctness in a debuggable environment, and testing its behavior at large scales in a distributed simulation. We have developed many distributed protocols and systems using WiDS, including a large-scale backup service that is robust enough to be deployed. We have also used WiDS to perform ultra-large scale (>1million instances) simulation of a production protocol. In this paper, we describe the principles and design of WiDS, share the lessons that we learned, and discuss on-going research that will further reduce programming and debugging difficulties of distributed systems.

1. Introduction

Research and development of distributed system has always been a tricky business. The process has many different stages, and each interdependent stage carries different requirements. The protocols must first be fully specified and proved. A correct implementation that follows is no trivial matter, as debugging a distributed system is a known hard problem. For the purpose of developing Internet-scale P2P systems [1][2][3], perhaps the most challenging is to fully understand any performance issues before the system is deployed.

To mitigate some of these difficulties, we find that a systematic approach is helpful. While the protocol specification, modeling, and proof remain too difficult to be incorporated in an integrated toolkit, we have united the rest of the processes in a single integrated toolkit called WiDS (*W*iDS *i*mplements *D*istributed *S*ystem).

The general philosophy of WiDS can be summarized as "code once and run many ways". WiDS adopts an object-oriented and event-driven programming model, and provides a small and straightforward set of APIs to support message exchanges and timers. Once a distributed protocol is developed, it can be simulated within a single address space on a single machine for debugging purposes, simulated on a cluster of machines to understand its macro-behavior, or deployed and run in the real. Users work with the same code base across different development stages and link it to appropriate libraries accordingly.

We have researched and developed many of our protocols and systems using WiDS, including a large scale, distributed backup service [4] that is robust enough to be deployed in MSR-Asia this year. We have also done extensive testing for production code of a P2P protocol [5] of more than one million instances, using hundreds of clustered PCs. To our knowledge, this is the largest P2P simulation that has ever been attempted. While all these exercises have demonstrated the value of such an integrated toolkit, our experiences also point out more challenging research directions to further reduce programming difficulties as well as to improve the debugging process.

Section 2 gives an overview of WiDS. We summarize our experience of performing complete system development and large-scale testing in Section 3. We discuss several new research focuses in Section 4. Section 5 discusses related work, and we conclude in Section 6.

† Work is done as intern in Microsoft Research Asia.

2. The WiDS Ecosystem

To serve as a generic ecosystem for distributed system development, WiDS needs to achieve several specific goals. First, there should be one single code base that is easily shared across different development stages. It is hazardous to maintain one code for simulation and another for real deployment, and try to sync up as progress is made. Second, while a distributed application is inherently more difficult to debug than a centralized one, we would like the users to spend their debugging energy in one address space as much as possible. Finally, when required, WiDS should support large-scale performance study for system scales approaching that of the real deployment.

Since a distributed system is essentially a collection of autonomous state machines, WiDS adopts an event-driven and object-oriented programming model, and is implemented using C++. A WiDS object represents a protocol instance or a service, and is identified by the tuple <WIDSNODE, WIDSSTUB>, analogous to how a networked service is addressed in the real world. WiDS objects exchange asynchronous messages to each other. Each message is dispatched to the target object's corresponding handler, which was declared using a macro. WiDS also provides periodic and one-time timers so that users can implement various failure detection mechanisms.

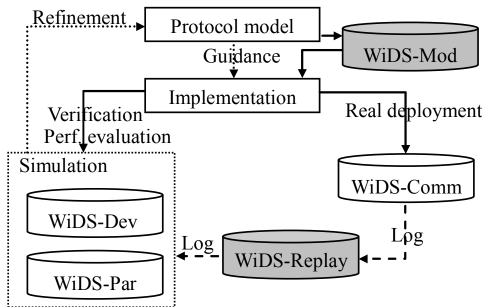

Figure 1. The ecosystem of WiDS-based protocol development and its five major components. The shaded ones (WiDS-Mod and WiDS-Replay) are under development.

These APIs isolate a WiDS-programmed protocol from any particular runtime that users want to employ. The WiDS runtimes fall into two general categories. The first is the *simulation mode*, where the runtime inserts and dispatches events through event wheel(s). Simulation mode supports pluggable topology models, allowing users to exercise different code paths in the protocol. The timestamp of a message is the source object's virtual clock plus the delay specified by the topology model. Eventwheel(s) ensure the chronological order of message execution, which in turn advances the simulation time. The simulation can be run on a single machine (linked with WiDS-Dev), enabling debugging of multiple instances of a protocol in the same address space. Alternatively, the simulation can be run in parallel on a cluster of machines to investigate performance issues for very large scales (linked with WiDS-Par). In the *network execution mode*, WiDS provides a socket-based library (WiDS-Comm), yielding a system ready to run in the real network environment. WiDS users always work with the same code base, invoking different runtimes by simply re-linking to different libraries according to their needs. Figure 1 summarizes these components of the WiDS development lifecycle. Two new members of the WiDS package, WiDS-Mod and WiDS-Replay, will be introduced in Section 4.

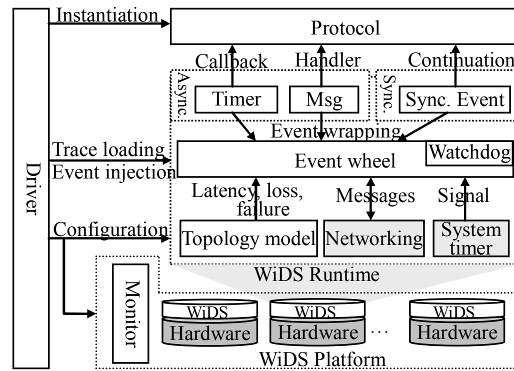

Figure 2. The WiDS architecture. Different runtimes are shaped by integrating some of the four modules: topology model, networking, system timer and event wheel.

Figure 2 depicts the WiDS runtimes. It contains topology models that generate latency and state for links between two simulated nodes, a *crystal* to trigger physical time signals, networking support based on native sockets to transport messages across physical machine boundaries, and an event wheel that stores all the events encapsulating messages, timers, and synchronous calls. Different WiDS runtimes are shaped by integrating some of these functionalities and, more importantly, different scheduling mechanism in the event wheel. There is a watchdog facility to check the progress of events, which is especially important to deal with stragglers in large-scale simulation. The monitor offers interactive simulation ability so that the user can break or step at event granularity. Along with the protocol, the user must also supply a driver program to instantiate the protocol instances, feed inputs, and inject events. In the simulation mode, the driver also specifies the topology model and node behavior (e.g., crash or create).

The WiDS parallel simulation is master-slave architected and proceeds in rounds. During each round, the master calculates a safe window (of simulation time) by looking at the head events of the slaves, and then informs the slaves to execute any events within that win-

dow. This barrier model becomes increasingly inefficient with more machines. To improve simulation performance, we have developed an optimization called Slow Message Relaxation (SMR) that simulates a window of ticks per round. This raises the possibility that a slave machine has already advanced its simulation clock when an event with a smaller timestamp arrives. We call such a message a *Slow Message,* and simply set its timestamp to the node's current clock value before passing it to its handler. The rationale is that this is as if the message had suffered some extra delay in the network. A correctly designed distributed protocol should have already handled any network-jitter generated abnormality. However, slow messages may lead to problems that otherwise would not have appeared such as false timeouts, and may change the statistics that the simulation is measuring as well. Our analysis shows that as long as the window width is kept under some value (automatically derived from the timer APIs), there will be negligible impact. Furthermore, the window width can be adaptively adjusted to achieve the optimal performance at runtime. We have verified that this optimization achieves an order of magnitude performance improvement simulating several large scale P2P protocols, without compromising the statistical accuracy of the simulation results.

3. Experience with WiDS

3.1 Complete system development

One of the complete distributed systems we have developed is the BitVault data retention platform [4]. BitVault employs commodity PCs as building blocks to construct a distributed backup service that is scalable, highly reliable, and highly available. Topology-wise, nodes are arranged in a ring. At the bottom layer, there is a voting-based failure detector to monitor the health of each node by a constant number of its neighbors. A failure or new node join event is then broadcasted in O(log$N$) steps to all other members, and anti-entropy is employed to ensure the eventual convergence of membership. These protocols comprise an eventual consistent membership protocol. Above that, a placement policy places multiple replicas on a constant number of nodes, and a distributed indexing mechanism tracks the location of an object. We use massively parallel repair to deliver order-of-minutes repair time for a failed disk upon the notification of membership change. There is a scalable monitoring infrastructure embedded in the system to trigger load balancing automatically. BitVault is entirely developed and maintained using WiDS. Each BitVault node comprises several objects that implement different functions (e.g., membership, monitoring, index, data etc.), and these objects communicate with each other using WiDS messages. BitVault is robust enough that we plan to roll out a 32-node installation as an interactive backup service in the first half of this year.

Although WiDS significantly improved the development process of BitVault, during the course we have learned several important lessons that lead to the further development and research focuses for WiDS. First, while the event-based programming model is a natural fit to implement state machines, it is still difficult to program and debug. This is especially true for protocols that have multiple phases. For those protocols, the event model will spread the protocol logic in multiple event handlers, and the program must therefore explicitly handle the context moving from one handler to the other. A protocol that is multi-phased but deals with a single remote party is most easily programmed using a single thread with remote procedure calls (RPC). However, the thread model falls short if the protocol has a concurrent phase that involves multiple parties, since it must spawn separate threads to deal with these parties and then sync-up later on. The thread model must also carefully guard critical sections, which is non-trivial and something that the event model does not need to handle. Many distributed protocols, however, are in fact both multi-phase and multi-partied (e.g., two-phase commit). A good number of BitVault protocols fall into this category. Therefore, in terms of programming effort, neither the event nor the thread model is an ideal fit. These experiences motivate us to develop both new APIs and architecture to further mitigate the program burden (c.f. Section-4.1).

Second, the WiDS runtime schedules at event granularity. This implies that events are handled in turn, and one's execution can not be preempted by others. It is usually not a problem. However, consider an event that is sandwiched by two heartbeat events. If the middle event takes an exceptionally long time to complete (e.g., a blocking disk I/O) then the timer logic can be violated. In the case of BitVault, it is possible for the failure detector to wrongly signal the crash of a node, allowing the repair mechanism to kick in, which can only make things worse. This particular issue can be resolved by offering a failure detection service inside the WiDS runtime so that one can register the interested endpoints and be notified when an endpoint fails to respond. By decoupling the dependency, the probe and response can run in parallel with the execution of events, fulfilled by the WiDS runtime. However, at its core, the issue is the handling of time-critical events and the provisioning of some level of real-time guarantee. Since objects typically implement a service (e.g., the membership protocol), and the WiDS objects communicate only through messages, one thing we plan to do is to allow events of

more time-critical objects to preempt other events. The other possibility is to develop a `Yield` API so that the user can chop a long-running event.

Third, related to the above two issues, many of the bugs did not manifest until the system was run in network execution mode, no matter how hard we tried to stress the code path in simulation mode. One reason is that event handling can take arbitrarily long in network execution mode, as opposed to one (simulated) clock tick in simulation. Thus the sequence of events can differ in unexpected ways, making it difficult to discover those bugs in the simulation environment. This experience propels us to develop WiDS-Replay (Section 4.2), which logs events and deterministically replays them in simulation mode. That is to say, we'd like to build a two-way street between WiDS-Dev and WiDS-Comm.

3.2 Large-scale testing

PNRP [5] is a P2P name resolution protocol with a target scale of tens of millions of nodes. Working with our product division partners, we ported PNRP to run on WiDS, and used WiDS-par to understand its macro-behavior. We have successfully completed many simulation runs of more than a million PNRP instances using hundreds of PCs. Some of the simulations took weeks to complete. This work has allowed us to gain insights into the system behavior, identify performance and network overhead, and remove design limitations that become apparent only under stress and at such a large scale.

Running a large-scale program on a cluster of machines almost inevitably brings up the same set of (mundane) issues. These include deploying and version-controlling the code, monitoring the health of the runs, managing the cluster, dealing with stragglers, and gathering statistics for final analysis. Moreover, heterogeneity in both software and hardware is much more than a performance (and hence configuration) issue. We ran into cases where some machines were equipped with mobile NICs or had stale network drivers and therefore could not handle bulk traffic. In both cases we ran micro-benchmark with a binary search strategy to isolate them. Clearly this process needs to be automated. Finally, we also realized that the master-slave architecture of WiDS-par needs to be changed if we are to attempt scales beyond a few million protocol instances.

Another approach we are considering is to swap states to disk and use intelligent prefetching policies to overlap the time of loading state from disk with simulation computation. By boosting per-machine simulation scale, we hope to reduce the number of total machines needed and thus the barrier overhead.

4. Research in Progress

4.1 WiDS-Mod

A typical development process starts with some pseudo-code that bridges protocol logic with the real implementation. Currently WiDS covers the development process starting from the implementation stage. The problem is that there is a large gap between the protocol logic and the final codes, resulting in coding as well as maintenance difficulties. This is especially problematic when there are many complicated and intertwined protocols involved in a system (as in BitVault).

WiDS-Mod borrows the principle of Intentional Programming [6] and adopts a hybrid approach. Taking advantage of temporal logic [7] and UML [8], our description language allows users to specify protocol logic in an abstract level and in the GUI (e.g. Figure 3(a)). The protocol logic is then automatically turned into skeleton code (c.f. Figure 3(c)). The users then fill in the rest of implementation, such as the code that examines the field of the `AckBuf` returned from the slaves to set the `all_ready` flag that decides whether to proceed to the commit phase of a two-phase commit protocol.

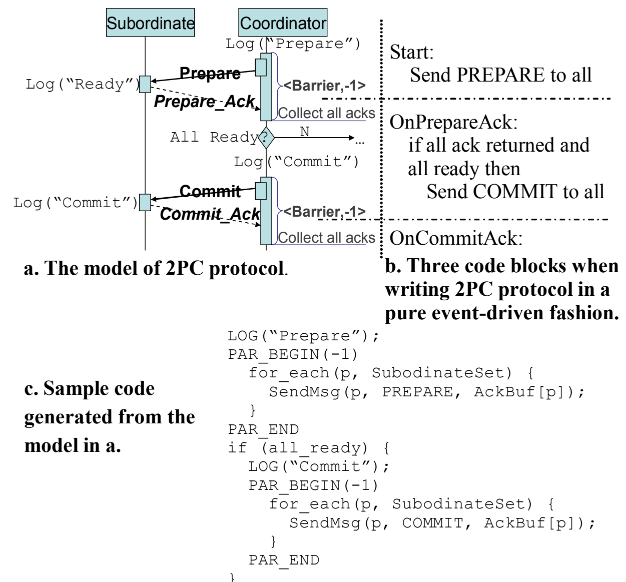

Figure 3. WiDS-Mod: (a) the model of the 2PC protocol; (b) codes using event-driven programming (coordinator side); (c) Sample code generated from the model.

This approach shrinks the gap between the high-level protocol specification and implementation, which is itself broken down into the logic level and the detailed handler level. Our point is that, for distributed system, these two levels already have inherently different natures and complexities (e.g., logical versus implementa-

tion correctness), so we might as well program them in different ways.

As we discussed in Section 3.1, many distributed protocols work in phases, each of which may involve multiple remote entities. A number of BitVault protocols fall into this category. For these protocols, a purely event-driven programming model quickly becomes awkward. Figure 3(a) shows the classic two-phase commit protocol, and the three separate code blocks (Figure 3(b)).

Independent of the modeling effort, therefore, we need to extend both the WiDS APIs and the runtime. For instance, `SendMsg()` is a synchronous call which will block the caller until the destination has processed the message and sent back acknowledgement, and `PAR_BEGIN/PAR_END` closure offers a barrier semantic, which will parallelize all synchronous messaging operations inside and resume execution when all of them are finished. With user-level threading [9], we will be able to wrap the synchronous calls in the continuation events and offer thread-like semantics, and can additionally accommodate multi-party semantics, something that the pure thread model has difficulty to do. All these attempts are to further reduce programming difficulties while leveraging the strengths of both the event and thread model.

4.2 WiDS-Replay

By exercising different network models, a good portion of protocol bugs can be rooted out. Unfortunately the remaining bugs, which will only surface in the network execution mode, are also the more difficult ones to find. In comparison, the cyclic debugging process [10] we are so used to in analyzing bugs in sequential applications, in which one sets a debugging point and repeats the execution, quickly becomes too much to afford. And yet writing out and then analyzing logs is also a grueling exercise. WiDS-Replay is a set of utilities aimed at analyzing bugs that occur only during the network execution by bridging with the simulation mode.

The general methodology of WiDS-Replay is straightforward. When running in the network execution mode, checkpoints are executed at each machine for all important states, and logs are also kept for any inputs between the checkpoints that may change the state of a running protocol instance (file I/O, wall-clock, random number generators, etc.). Finally, user-defined logs are coalesced and dumped into the same log file. We then start the protocol in the simulation mode, reloading the checkpoint and log traces to reconstruct context. Notice that in the network mode every instance is running as a separate process, whereas in the simulation mode each instance is a WiDS object. Therefore we carefully perform data marshalling and de-marshaling to make sure that the object states are loaded correctly. In the replay phase, we navigate the traces at the granularity of log entries while bringing up the code alongside as the navigation context. We then use deterministic forward and backward replay to examine the program state, doing this across different objects (and hence protocol instances running on different machines) when necessary.

The object-oriented programming model of WiDS makes it possible to replay a distributed protocol within a single address space and on a single machine. Therefore, WiDS-Replay provides the capability of *virtualizing* the distributed system debugging process. A prototype of WiDS-Replay has already been built, but much more needs to be researched and developed before it can be put into practice.

WiDS-Replay can also work within the simulation mode. Here, periodic checkpoint is sufficient for deterministic replay, assuming that the simulation environment is also checkpointed. One may argue that since simulation is deterministic anyways, why bother with checkpointing. The truth is that for a complex protocol, it often takes a long time to reach a faulty point. Checkpointing segments the debugging process and allows the user to invoke different debugging details when appropriate.

5. Related Work

As observed in [11], sharing the same code base for the purpose of development, simulation, and deployment is a popular notion. There have been some attempts along the same line. For instance, Neko [12] is a java platform that allows the same algorithm to run both in simulation and in real network. Though we do share the same philosophy, their interfaces and architecture are quite different from ours. Neko does not offer parallel simulation capability, and it is not clear whether it has been used to build a complete system. While WiDS offers native C/C++ support, MACEDON [13] takes a different approach by offering a domain-specific language for FSM (finite state machine) based protocols. The MACEDON approach is geared towards quick prototyping overlay applications. Large-scale performance study requires an emulation approach (discussed below), though it should be possible to add PDES (Parallel Discrete Event Simulation [14]) support as well. One thing that MACEDON does very well is to abstract many common services of overlay systems into generic packages. WiDS can take the same approach for services such as failure detector and membership protocols, which are common building blocks for distributed system.

One contribution of the current WiDS package is its capability of performing large-scale simulation and testing on clustered machines. While there have been many works on PDES, our Slow Message Relaxation optimization is unique in that it takes advantages of the time slacks that all distributed protocols use to cope with unreliable network transmission. A related approach to large scale testing is *emulation*, which is exactly the same as the network execution mode of WiDS except that many (typically thousands of) instances of protocols run on each testing node, and the packets are routed through a cluster of machines modeling the Internet topology and (therefore) packet delays [15][16]. There are pros and cons in these two approaches, and it will be an interesting research topic to identify synergy.

The versatility of WiDS extends to cover other important aspects of the development process. WiDS-Mod borrows principles from Intentional Programming [6] to abstract high level logic (intention) from implementation. WiDS-Mod provides a natural and formal model, and yet reserves sufficient flexibility for developers to describe their implementation details.

The idea of using checkpoint and logging at runtime to discover difficult bugs using deterministic replay is an old one [17]. WiDS-Replay checkpoints and logs distributed protocols as they are run in the real environment, but deterministically replays and debugs the protocols on a single machine within one address space. As far as we know, this is a novel approach.

6. Conclusion

WiDS was born in response to many early lessons we learned when researching and developing several P2P protocols. As an integrated toolkit that covers rapid prototyping, large-scale simulation, and deployment, it has already significantly improved our productivity. Still, to become truly holistic, WiDS must evolve further to address the difficulties of programming as well as debugging distributed systems.

Acknowledgement

We would like to thank Noah Horton, Geogy Samuel, Brian Lieuallen and Sandeep Singhal for the support of running large-scale simulation of the PNRP protocols using WiDS. We also thank the anonymous reviewers and Richard Draves and Kurt Akeley for their valuable inputs.

Reference:

[1] P. Druschel and A. Rowstron, “PAST: A large-scale, persistent peer-to-peer storage utility”, in Proc. HotOS VIII, Schloss Elmau, Germany, May 2001

[2] John Kubiatowicz, David Bindel etc., “OceanStore: An Architecture for Global-Scale Persistent Storage”, in Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000

[3] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, “Ivy: A Read/Write Peer-to-peer File System”, in OSDI, December 2002

[4] Zheng Zhang, Qiao Lian, Shiding Lin, Wei Chen, Yu Chen, Chao Jin, “BitVault: a Highly Reliable Distributed Data Retention Platform”, under submission

[5] Microsoft TechNet, “Introduction to Windows Peer-to-Peer Networking”, http://www.microsoft.com/technet /prodtechnol/winxppro/deploy/p2pintro.mspx

[6] C. Simonyi. “The Death of Computer Languages, The Birth of Intentional Programming”, Technical Report MSR-TR-95-52, Microsoft Research, 1995

[7] Leslie Lamport, “Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers”. Addision-Wesley (2002).

[8] Unified Modeling Language 1.5, OMG, http://www.omg.org/technology/documents/formal/uml.htm

[9] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer, “Capriccio: Scalable Threads for Internet Services”, In Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP), 2003.

[10] Joel Huselius. “Debugging parallel systems: A state of the art report”. Technical Report 63, Mlardalen University, Department of Computer Science and Engineering, September 2002

[11] Michael Jones and John Dunagan, “Engineering Realities of Building a Working Peer-to-Peer System”, MSR Technical Report MSR-TR-2004-54. June 2004.

[12] P. Urban, X. Defago, and A. Schiper, “Neko: A single environment to simulate and prototype distributed algorithms”, in Proc. of the 15th Int'l Conf. on Information Networking (ICOIN-15), (Beppu City, Japan), Feb. 2001.

[13] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat, “MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks”, in Proc. of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI), 2004.

[14] A. Ferscha, and S.K. Tripathi, “Parallel and distributed simulation of discrete event systems”. Technical report, University of Maryland, August 1994.

[15] Amin Vahdat, Ken Yocum, Kevin Walsh, etc., “Scalability and Accuracy in a Large-Scale Network Emulator”, in OSDI, December 2002

[16] Emulab project, the Utah network testbed (Web site). http://www.emulab.net/.

[17] E. N. M. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson, “A survey of rollback-recovery protocols in message-passing systems”, ACM Computing Surveys (CSUR), 34(3):375-408, 2002