Emulation of Multi-Hop Wireless Ad Hoc Networks

Qifa Ke † David Maltz † David B. Johnson ‡
† School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
E-mail: ke+, dmaltz@cs.cmu.edu
‡ Department of Computer Science, Rice University, Houston, TX
E-mail: dbj@cs.rice.edu

Abstract

There are two usual methods to evaluate a software system in multi-hop wireless ad hoc networks: simulation and real test-bed. The test-bed method is expensive and non-repeatable. The simulation method usually requires re-implementing the real software system inside the simulator, which is also infeasible for large scale software systems. In this paper, we present an emulation system capable of evaluating unmodified real software systems in simulated environments, which is repeatable, detailed, and realistic. The experimental results show that our system is able to emulate large scale ad hoc networks. By using our system, we have greatly improve the performance of the Coda file system in ad hoc networks.

1 Introduction

An ad hoc network is a collection of wireless mobile nodes which dynamically forms a temporary network without using the existing network infrastructure or centralized administration. Due to its dynamic topology and limited resource, new network protocols and applications have been developed specifically for ad hoc networks. It is very important but nontrivial to evaluate network protocols and applications in ad hoc network environments. Usually the following two methods are used:

- The first method is to construct a real ad hoc network test-bed with desired scenarios, and then run the applications or protocols in the test-bed. Although the scenario is very realistic, this method is expensive and non-repeatable.

- The second method is using network simulator. It offers the ability to repeat and control the network conditions at user's requirement. One example is the network simulator *ns-2* [5]. However, pure network simulation requires re-implementing the network protocols/applications inside *ns-2*. Some systems may be too complicated to be re-implemented, such as the Coda file system [10]. Also, the traffic used in the simulator is generated by traffic model. Traffic modeling is nontrival.

Since both test-bed and simulator have pros and cons, network emulation [4] has been proposed to combine their advantages. A network emulator is a tradeoff between real testbed and pure simulation. In a typical emulation experiment, network traffic is generated by real systems and then injected into the simulator to experience the simulated network environments. By allowing the real world traffic to interact with the simulator, network emulator avoids the traffic modeling problem, as well as the problem of re-implementing the real systems inside the simulator. At the same time, it achieves the repeatability by simulating the network environments inside the emulator.

The emulator developed by Kevin Fall [4] is mainly for wired networks. It does not contain layers below TCP. However, these lower layers are important in constructing realistic ad hoc network environments. Based on the CMU wireless extension to *ns-2* [3], we have implemented a detailed and realistic emulation system which contains all of the necessary network layers. Therefore, without implementing the real system in *ns-2* and without deploying and operating physical machines in the field, we are able to evaluate unmodified real systems under realistic and repeatable ad hoc network conditions.

In order to correctly measure the real system's performance, the emulator needs to run in real time, which requires the virtual clock in the simulator be synchronized with the wall clock. The validity of the emulation experiments depends on the emulator's ability to keep up with the real time. We have developed a re-ordering algorithm to improve this ability. The improved emulation system is able to emulate ad hoc networks containing more than 100 mobile nodes with heavy background traffic.

We apply our emulator to analyze and improve the Coda [10] file system over ad hoc networks, and achieve substantial improvements.

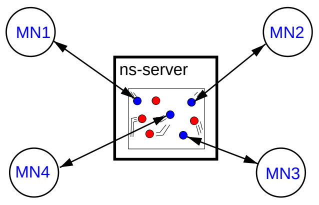

Figure 1: Logical view of emulation setup. MNi is a real machine which will generate real traffic. This traffic is directed to its corresponding emulated node inside the ns-server, which is part of the ad hoc network scenario.

2 Structure of the Emulator for Ad Hoc Networks

In this section, we briefly review the existing emulator in *ns-2* and then present our emulator for ad hoc networks.

2.1 Emulator in *ns-2*

The network emulator in *ns-2* [4] is a real-time simulator with the ability to interact with real-world traffic. To achieve this function, the following three major components have been added to *ns-2* [4]:

- A *real-time scheduler* to synchronize the simulator's virtual clock with the wall clock.
- *Network objects* to access the live traffic. Three types of network objects are implemented in [4]. They are UDP/IP, raw IP, and frame level network objects.
- *Tap agents* to tunnel live network traffic inside the simulator.

2.2 Emulator for Ad Hoc Networks

The network emulator [4] in *ns-2* is mainly for wired network, where protocol layers below TCP are not emulated. These lower layers are important for constructing realistic ad hoc network environments. With the new elements added by CMU wireless extensions, we have designed an emulator for multi-hop wireless ad hoc networks, where real world traffic can experience a realistic ad hoc network environment.

Figure 1 shows the logical view of the set up for emulation of ad hoc networks, where the **ns-server** is a single central machine running *ns-2* simulator, and other machines *MN1,..., MN4* are real machines running real network protocols/applications.

In the ns-server, users create ad hoc network scenarios[1] for their desired ad hoc network environments In our emulation system, we design the following two types of mobile nodes for the ns-server:

[1]Ad hoc key [1] is a convenient tool to create scenarios of wireless networks.

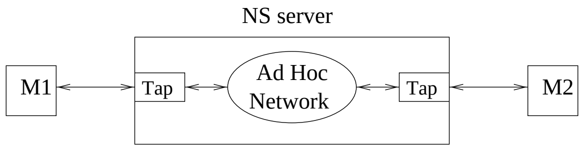

Figure 2: Packet flow in the emulation system, where M1 and M2 are real computers in which the real applications/protocols are running.

- A *simulated mobile node* is a usual mobile node in *ns-2*. It contains a protocol stack from physical layer to application layer.
- An *emulated mobile node* is a simulated mobile node augmented with a *tap agent* and a *network object*. Each emulated node is tightly bound with a real machine where real protocols/applications under investigation are running. Traffic generated by the real machine is directly captured by the corresponding emulated node inside ns-server, which is part of the ad hoc network scenario.

In Figure 1, all of the traffic generated by *MNi* is directed to its corresponding emulated node.

Although the real machine *MNi* stays static, its corresponding emulated node is one of the mobile nodes in the simulated ad hoc network. All of the traffic generated or sunk by *MNi* is relayed by its corresponding emulated node. Therefore, we achieve the effect that the real machine *MNi* is a mobile node in the ad hoc network, with its real traffic undergoing the protocol stack and experiencing the ad hoc network environments created by user.

3 A New Real-Time Scheduler Using Reordering Algorithm

In this section, we present a new real-time scheduler using reordering algorithm, which reduces the time-lag (if an event is scheduled at time t_1 and is processed at time t_2, then its time-lag is $t_2 - t_1$) of visible events, thus improves the accuracy and scalability of emulation.

3.1 Visible Event v.s. Invisible Event

We identify the difference among events by classifying them into two categories: *visible event* and *invisible event*. In Figure 2, a data packet from M1 to M2 will take the following path:

1. generated by the real system running in real machine M1 and then sent to its destination M2;
2. tapped by the ns-server and injected into simulator by the tap agent in the emulated node representing M1;

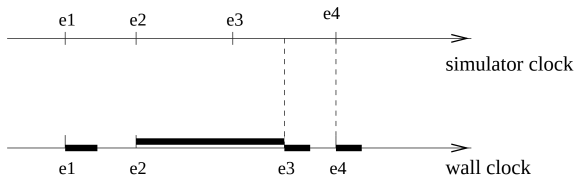

Figure 3: Invisible time-lag. Thick line in the axis of wall clock indicates the processing time of each event.

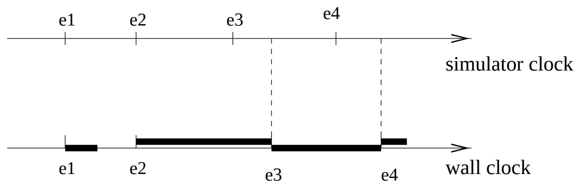

Figure 4: Visible time-lag. Thick line in the axis of wall clock indicates the processing time of each event.

3. experiences the ad hoc network environments simulated in *ns-2*;

4. sent to the real destination M2 by the tap agent in the emulated node representing M2;

5. received by the real destination M2;

The extra overhead caused by the emulation comes from step 2, step 4, and the time for transmitting the packet between real machines and ns-server [2]. The events happen at step 2 and step 4 are *visible events*, which only happen in emulated nodes. Other events are not visible by the real machines and are called *invisible events*. If the visible events are processed without any delay, then the real systems running in M1 and M2 will not experience any delay either, and their behaviors are preserved in the emulator.

We give two examples to show the different effects of the time-lag of visible/invisible events. In Figure 3 and Figure 4, $e_i(i = 1, ..., 4)$ is an event in the event queue, whose scheduled time is indicated by the position in the axis of the simulator clock. The actual dispatching time of e_i is indicated by the position in the axis of real time clock, where the thick lines indicate the processing time of each event. We assume that e_4 is a visible event, and other events are all invisible events in simulated nodes.

- In Figure 3, e_3 has time-lag due to the process time of e_2. However, the visible event e_4 is dispatched on time. The real machine will not be aware of the time-lag of e_3. And the behavior of the application running in the real machine will not be changed due to the time-lag of invisible event e_3.

[2]This is ignored since it only takes about 80 μs to transmit a 1 KB packet in a 100 Mbps switcher used in our test-bed.

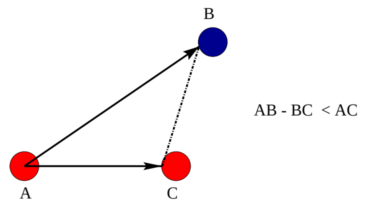

Figure 5: An example scenario where reordering algorithm will take advantages by dispatching the visible events in advance of their scheduled times. In this scenario, B is an emulated node, and A, C are simulated nodes. B's nearest neighbor is C and $|AB| > |AC|$. When A sends out a packet, it arrives at C first and is processed there first by the simulator. Using reordering algorithm, since $|AB| - |BC| < |AC|$, the packet will be processed at the emulated node B first instead of C.

- In Figure 4 both e_3 and e_4 have time-lag. These time-lags will affect the behaviors of the real application. However, the effect on the application is only determined by the amount of time-lag of visible event e_4. The time-lag of e_3 is still invisible to the real application.

From above we can see that only the time-lags of visible events can affect the real applications, which leads us to develop a new real time scheduler using re-ordering algorithm.

3.2 Reordering Algorithm

Since only the time-lags of visible events can affect the real applications, we can improve the emulator by dispatching the visible event ahead of its scheduled time if allowed, which may result in dispatching events out of their schedule-time orders.

Dispatching events out of schedule-time orders may cause *causality error*. For example, if we dispatch event e_{t2} before e_{t1}, with scheduled time $t_2 > t_1$, and if e_{t2} changes some state variables which will be used by e_{t1}, then we will have the effect that future event e_{t2} affects the past event e_{t1}, which is called causality error. However, if these two events do not have causality relationship, dispatching them out of time-stamp orders will still preserve the correctness of the simulated model.

We have proved the following *reordering algorithm*[6]

Reordering algorithm: In a multi-hop wireless ad hoc network, an event E of Node N scheduled at time t can be dispatched at time $max\{t-\triangle t, t_N\}$, where t_N denotes the latest dispatching time of Node N, and $\triangle t$, the possible maximum time that an event can be advanced, is determined by:

$$\triangle t = \frac{\text{distance\_from\_nearest\_neighbor}}{\text{speed\_of\_light}} \quad (1)$$

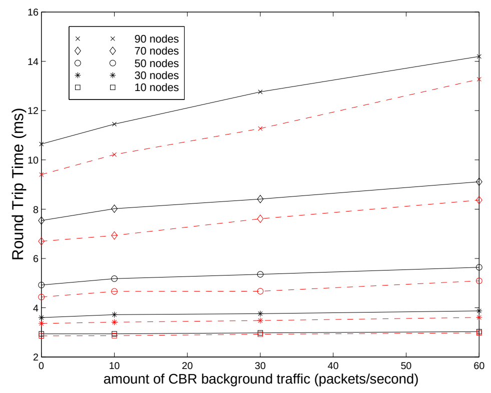

Figure 6: Round-trip time of ICMP echo request, where dashed lines are the results of reordering algorithm, and solid lines are the results of non-reordering algorithm.

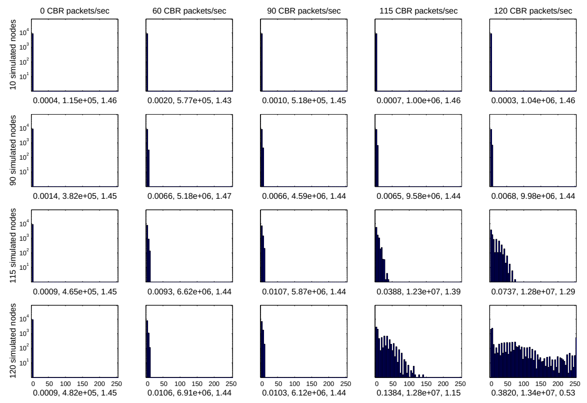

Figure 7: The histogram of visible time-lags. The three numbers under each plot are the maximum time-lag (ms), total number of events, and the real FTP throughput (Mb/second) respectively
.

Intuitively, the upper limit of $\triangle t$ is derived from the fact that a neighbor of node A takes at least $\triangle t$ to put wireless energy on machine A. Therefore, an event scheduled at time t can be safely dispatched at time $max\{t - \triangle t, t_N\}$. If we apply the reordering algorithms to the events belonging to emulated nodes, then we can process the visible events in a higher priority. Although $\triangle t$ is very small, bursts of simulated events can happen during $\triangle t$ in a wireless network with shared channel, especially in lower level layers (physical layer, MAC layer, and routing layer). For example, when a node sends a packet, it schedules a packet-arrive-event at every node sharing the same channel. Figure 5 shows one example that reordering algorithm will take advantages of $\triangle t$.

To illustrate the improvement achieved by the reordering algorithm, we measure the average round-trip time of ICMP echo requests between two emulated nodes, which are in a cloud of mobile nodes. Figure 6 shows the results, where dashed lines are the result of reordering algorithm, and solid lines are the result of non-reordering algorithm. As we can see from this figure, reordering algorithm has smaller RTT by reducing time-lags of visible events. The improvement for typical scenarios ranges from 5% to 10%. In a scenario with larger number of nodes, there are more chances where reordering algorithm can take advantages.

4 Scalability of the Emulator for Ad Hoc Networks

With large scale complex simulation scenarios, the emulator may not be able to keep up with the real time, which in turn will change the behaviors of the real system under evaluation, and invalidate the emulation experiment. This section examines the scalability of the emulator.

To evaluate the scalability of the emulator, we vary the load of the ns-server by changing the complexity of simulation scenarios, which is determined by the number of nodes and the amount of background traffic. The real application we use here is FTP. In our experiments, the number of mobile nodes varies from 10 to 120, and the rate of background traffic from 0 packets/second to 120 packets/second (the packet size is 512 bytes).

Figure 7 shows the experimental results. In this figure, each plot shows the histogram of visible time-lags in its corresponding scenario. The horizontal axis represents the amount of time-lags in *ms*, while the vertical axis shows the number of time-lags. The three numbers under each plot represents respectively the *maximum time-lag (ms)*, the *total number of events*, and the *throughput (Mb/second)* of real FTP. As the scenario complexity increases, the number and the amount of time-lags increase, and the FTP throughput decreases, eventually results in an invalidated emulation experiment with a large scale scenario of 120 nodes with a background traffic of 120 packets/second. In this large scale scenario, there are many large visible time-lags, and the FTP throughput is only 0.53 Mb/s, which is very small compared to 1.46 Mb/s, the throughput of simulated FTP for the same scenario. FTP/TCP is well modeled in *ns-2*, so we can use it to judge the validity of emulation experiments on FTP/TCP. In cases that real system is not modeled in the simulator, we can use the statistics of the time-lags of visible events to determine the validity of each emulation experiments. The four experiments in the bottom-right corner of Figure 7 are either invalidated or questionable. All other experiments have small time-lags and throughput closed to 1.46 Mb/s, and are therefore valid.

As we can see from this figure, the emulation system works for ad hoc networks with sufficient size and complexity to enable studying and evaluating real system under interesting scenarios of ad hoc net-

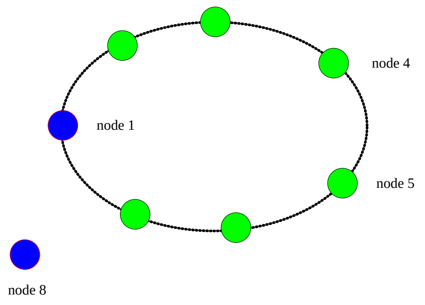

Figure 8: Scenario for testing Coda file system, where node 1 fetches data from node 8. Node 8 is a stationary node, and all the others move along the circle at constant speed. This scenario is interesting since the topology keeps changing from the Node 8's point of view, with its route to node 1 keeps changing.

works.

5 Improve the Coda File System in Ad Hoc Networks

Coda file system [10] is the descendant of Andrew file system (AFS). It is a distributed file system consisting of file servers and clients. Evaluating Coda file system under ad hoc network environments has been interesting but have not been done before due to the difficulty of constructing real and repeatable ad hoc network environments. Also it is very difficult to model the Coda file system in the simulator since Coda contains more than 100K lines of code and has been tuned for years. In this section we apply the emulation system to analysis and improve Coda under ad hoc networks.

Here we give a brief review of the the file transfer protocol in Coda, which is essentially a cyclic blast transfer protocol. At the beginning of each cycle, the source sends a block of data packets (the last packet of each block has *ack-me* flag), then waits for the acknowledgement. The acknowledgements will specify the lost packets that need to be retransmitted. Then the cycle repeats. Note that in Coda, the client and server are not peers. Server will handle all Coda flow control regardless it is the source or the sink, so that a server can handle larger number of file transfer requests from clients. In other words, if the source is a server, it will retransmit the block of data if the acknowledgements do not arrive after a predetermined time (i.e., when the retransmission timer is timeout). However, if the source is a client, the client will wait passively for acknowledgements to arrive from server. There is not timeout scheme built in the client. Instead, the server will retransmit the acknowledgement if it does not receive more data packets from the client after a predetermined period of time. The retransmission timer is set based on the RTT estimation.

Ad hoc networks provide new environments with dynamical network topologies and error-prone low-bandwidth wireless channels, where the original Coda file transfer protocol does not perform as well as in other types of networks, probably due to the fact that some weakness in Coda may not exhibit in scenarios other than ad hoc networks. In emulator, since we can create rich accurate and realistic ad hoc network scenarios, quite a few problems in Coda are identified and fixed. For example:

1. In Coda, the last packet in one transfer window had the *ack-me* flag, and the receiving side will not send out the ACK packet until it receives this last packet. In ad hoc networks, this last packet is the most likely packet to get lost due to congestion or route error. We have observed that the first a few packets in one transfer window are usually transferred successfully after a route between the source node and the destination is established. Then route error may happen due to mobility or other reasons, and the last one or more packets in the transfer window are more likely to loss. To fix this problem, the receiving side now sends out a gratuitous ACK whenever it has seen a nearly full window of data packets, instead of passively waiting for the last packet in that window.

2. If the sending side was time-out while waiting for acknowledgements, Coda starts retransmitting all unacknowledged packets. In ad hoc networks, it is more difficult to choose a good time-out value. Also the ACK packet has much higher lost rate than in other type of networks. If the time-out value is not good enough or the ACK is lost, then retransmitting all unacknowledged packets wastes the bandwidth and increases the congestion, since some of the unacknowledged packets may have been successfully received by the receiver. To fix it, now we only retransmit the first unacknowledged packet (with *ack-me* flag) plus the next sequence of data packets that are allowed to send in the current send window. The retransmit packet usually triggers another ACK packet from the receiving side, which specifies the packets that are missed.

3. In Coda, when the transfer begins, the RTT is estimated and the retransmit interval (time-out value) is set according to the estimated RTT. This retransmit interval is fixed during that specific file transfer. While fixed retransmit interval may be good enough for other networks, ad hoc networks exhibit much larger RTT variance, which means fixed retransmit interval may result in unnecessary retransmit, or even worse, putting the link in a idle state for long time if the initial RTT estimated is too large. To fix

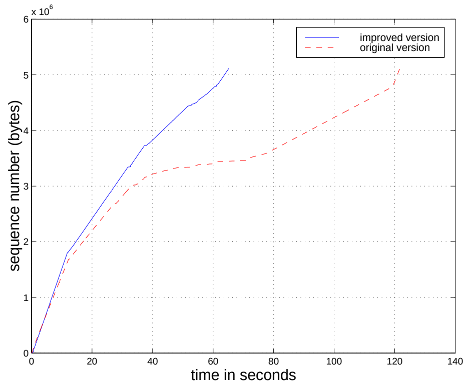

Figure 9: Sequence plot. The throughput of improved Coda file transfer protocol is almost twice of the original throughput.

this problem, the retransmit interval is now adjusted according to the estimated RTT.

Figure 9 illustrates our improvements on Coda over ad hoc networks. The scenario we used here is shown in Figure 8, where node 1 fetches 5M byte data from node 8, and there is a CBR background traffic (20 packets/second) between node 4 and 5. In this emulation experiment, all of the visible time-lags are less than 1.6 ms, so the emulation experimental results are validated. As we can see from Figure 9, the throughput of the improved version is almost twice of the original version's throughput.

6 Related Work

Besides the basis emulation work by Kevin Fall [4], other recent works in emulation include [2, 7, 9]. These systems are implemented by extending the kernel to intercept and drop packets at IP layer. The kernel implementation limits the usage of those systems, especially for the case of ad hoc networks, where simple packet manipulations inside kernel is difficult to model ad hoc networks.

The other closely related work is trace-based network emulation [8], which re-creates the observed end-to-end characteristics of a real network trace by using probing traffic. The created trace can then be replayed easily for evaluating real systems. Trace-based network emulation has been successfully used in analyzing system such as Coda in wired or wireless networks with infrastructure, where trace are relatively easy to collected. Still it could be very difficult to collect a trace of large scale ad hoc networks.

7 Conclusion

In this paper, we have presented an emulator for ad hoc networks which combines the strengths of both test-beds and pure simulation.

By differentiating visible events from invisible events, we also introduce a reordering algorithm, which processes the visible events ahead of their scheduled time. Since the real systems are only affected by the visible events, the reordering algorithm reduces the time-lags of visible events, therefore reduce the bad effects caused by the time-lags in ns-server. The scalability test has shown that the emulator works for ad hoc networks with sufficient size and complexity.

The improvement on Coda over ad hoc networks has demonstrated that our emulation system is a new powerful tool in analyzing and evaluation real systems in ad hoc networks. The advantages of using emulator for ad hoc networks are:

- We do not need to modify the real systems or model them in the simulator.
- The ad hoc network environments are under user's control. They are repeatable, detailed, and realistic. We do not need to drive the mobile nodes to construct a realistic test-bed.
- The validity of the emulation experiments can be determined by the time-lags of the visible events, which can be easily collected.

Acknowledgements
We would like to thank Jan Harkes for his help in testing and evaluating the SFTP in Coda.

References

[1] *The CMU Monarch Project's ad-hockey Visualization Tool for ns Scenario and Trace Files*, November 1998. Available from http://www.monarch.cs.cmu.edu/cmu-ns.html.

[2] J. Ahn, P.B. Danzing, Z. Liu, and L. Yan. Evaluation of TCP Vegas: Emulation and Experiment. In *Proceedings of the ACM SIGCOMM*, pages 185–195, Aug. 1995.

[3] Josh Broch, David A. Maltz, David B. Johnson, Yih-chun Hu, and Jorjeta Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proceedings of the ACM/IEEE MobiCom'98*, pages 85–97, Oct 1998.

[4] Kevin Fall. Network Emulation in the VINT/ns Simulator. In *Proceedings of the Fourth IEEE Symposium on Computers and Communications (ISCC'99)*, July 1999.

[5] Kevin Fall and Kannan Varadhan, editors. *ns* Notes and Documentation. The VINT Project, UC Berkeley, LBL, USC/ISI, and Xerox PARC, January 1999. Available from http://www-mash.cs.berkeley.edu/ns/.

[6] Qifa Ke. *Proof of the Reordering Algorithm in the Emulator for Ad Hoc Networks*. Avaiable from http://www.cs.cmu.edu/~ke/prove.ps.gz.

[7] National Institute of Standards and Technology. *Nistnet*. Avaiable at http://snad.ncsl.nist.gov/itg/nistnet.

[8] Brian Noble, M. Satyanarayanan, Giao Nguyen, and Randy Katz. Trace-Based Mobile Network Emulation. In *Proceedings of ACM SIGCOMM'97*, 1997.

[9] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1), Jan. 1997.

[10] M. Satyanarayanan. Coda: A Highly Available File System for A Distributed Workstation Environment. In *Proceedings of the Second IEEE Workshop on Workstation Operationg Systems*, Pacific Grove, CA, September 1989.