

Implementation and Evaluation of Secure Routing Primitives

Atul Singh Tsuen-Wan “Johnny” Ngan Peter Druschel Dan S. Wallach

Department of Computer Science, Rice University
{atuls, twngan, druschel, dwallach}@cs.rice.edu

Abstract

This paper provides details on practical implementation and optimizations of constrained routing table in Pastry [5] and the secure routing primitive as described in Castro et al. [1].

1 Introduction

In an overlay network, each node maintains links to a relatively small set of neighboring nodes. All communication within the overlay, be it related to maintaining the overlay or to application processing, occurs on these links. The operation of the overlay depends on the ability of correct nodes to communicate with each other over a sequence of overlay links. In an Eclipse attack [1, 7], a modest number of malicious nodes conspire to fool correct nodes into adopting the malicious nodes as their neighbors, with the goal of dominating the neighbor sets of correct nodes. If successful, an Eclipse attack enables the attacker to mediate most overlay traffic and effectively “eclipse” correct nodes from each others view. In the extreme, an Eclipse attack allows the attacker to control all overlay traffic, enabling him to deny service or censor content at will.

Castro et al. identify the Eclipse attack as a threat in overlay networks [1]. To defend against this attack, they propose the use of *Constrained Routing Tables* (CRT), where they impose strong structural constraints on the neighbor set. An idealized CRT guarantees that the expected fraction of malicious nodes in the neighbor set of correct nodes would be the same as the total fraction of malicious nodes in the network. However, practical implementation details, as well as measurements of the actual cost, are not available. In this paper, we describe our implementation of the proposed secure routing primitives in Pastry [5] and present simulation results to evaluate its effectiveness and overhead.

This paper is organized as follows. Section 2 provides background information on Pastry routing and secure routing primitives. In Section 3, we describe our implementation, including optimizations to reduce the overhead. Section 4 describes how an adversary can mount an Eclipse attack by manipulating the protocol and its impacts. Simulation results of our implementation are provided in Section 5. Section 6 concludes.

2 Secure Routing Basics

This section summarizes the the necessary background to understand the paper. It includes a discussion of Pastry [5], proximity neighbor selection (PNS) [2], and secure routing primitives [1].

2.1 Pastry

Pastry is a scalable, self-organizing structured peer-to-peer overlay network similar to CAN [4], Chord [8], and Tapestry [9]. In Pastry, nodes are assigned random identifiers (called nodeIds) from a large id space. NodeIds are 128 bits long and can be thought of as a sequence of digits in base 2^b (b is a configuration parameter with a typical value of 4). Given a message and a key, Pastry routes the message to the node with the nodeId that is numerically closest to the key, which is called the key's root. This simple capability can be used to build higher-level services like a distributed hash table (DHT) or an application-level group communication system like Scribe [6].

In order to route messages, each node maintains a routing table and a leaf set. For a network of N nodes, a node's routing table has about $\log_{2^b} N$ rows and 2^b columns. The entries in row r of the routing table refer to nodes whose nodeIds share the first r digits with the local node's nodeId. The (r, c) slot in the routing table contains a node whose $(r + 1)^{\text{th}}$ digit equals c . At each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node's nodeId but is numerically closer.

Each Pastry node maintains a set of neighboring nodes in the nodeId space (called the leaf set), both to ensure reliable message delivery, and to store replicas of objects for fault tolerance. The expected number of routing hops is less than $\log_{2^b} N$. The Pastry overlay construction incorporates proximity in the underlying Internet. Each routing table entry is chosen to refer to a node with low network delay, among all nodes with an appropriate nodeId prefix. As a result, one can show that Pastry routes have a low delay penalty: the average delay of Pastry messages is less than twice the IP delay between source and destination [2]. Similarly, one can show the local route convergence of Pastry routes: the routes of messages sent to the same key from nearby nodes in the underlying Internet tend to converge at a nearby intermediate node. A full description of Pastry can be found in [5].

2.2 Proximity Neighbor Selection

Proximity Neighbor Selection (PNS) constructs a topology-aware overlay by choosing routing table entries to refer to the topologically nearest nodes among all nodes with nodeId in the desired portion of the id space. The effectiveness of this technique depends on the degree of freedom an overlay protocol has in choosing routing table entries without sacrificing the expected number of routing hops. In prefix-based protocols like Pastry, the upper levels of the routing table have high flexibility, with the flexibility decreasing exponentially down the lower levels. As a result, the expected delay of the first hop is very low, increases exponentially with each hop, and the delay of the final hop dominates the whole path. Castro et al. [2] show that selecting routing table entries in this way leads to low delay stretch and significantly reduce the overlay routing overhead by avoiding wide area paths in favor of local paths. Their analysis and simulations confirm that proximity neighbor selection yields good performance at very low overhead.

2.3 Secure Routing Primitives

The secure routing primitive ensures that when a non-faulty node sends a message to a key k , the message reaches all non-faulty members in the set of replica roots R_k with a very high probability. R_k is defined as

the set of nodes that contains, for each member of the set of replica keys associated with k , a live root node that is responsible for that replica key. In Pastry, for instance, R_k is simply a set of live nodes with nodeIds numerically closest to the key. Secure routing ensures that (1) the message is eventually delivered, despite nodes that may corrupt, drop, or misroute the message; and (2) the message is delivered to all legitimate replicas for the key, despite nodes that may attempt to impersonate a replica root.

2.3.1 Redundant Routing

Redundant routing is a mechanism that uses redundancy to improve the probability of successful routing. As proposed by Castro et al. [1], redundant routing consists of the following steps: (1) use ℓ redundant routes in the overlay to reach nodes with identifiers in the vicinity of a desired point X in the id space; (2) any node receiving the above message and has X in its vicinity responds; (3) take a union of all nodes who responded and then ask each member in this union set to respond with *AGREE* if the union is their view of correct replica set, otherwise forward the message to remaining members and respond with *DISAGREE*; and (4) if everyone responded with *AGREE* in previous phase, the computed union is the required replica set and return; else repeat from (1) for up to 3 times.

2.3.2 Routing Failure Test

It is necessary to have a scheme for a node to estimate whether routing has been performed successfully. Since nodeIds are assumed to be uniformly distributed, the routing failure test is based on the observation that if faulty nodes try to suppress the existence of some correct nodes, the density of nodeIds in the id space would be much lower than the average. The test works by comparing the density of nodeIds in the neighbor set of the sender with the density of nodeIds close to the replica roots of the destination key. If the density is suspiciously low, the secure routing is repeated with a different set of routes. Specifically, let the nodeId density around the responding node be δ and the local nodeId density be α . We accept the responding node only if $\delta \leq \alpha\gamma$, where γ is a parameter chosen for minimizing false positives and false negatives.

2.3.3 Constrained Routing Tables (CRT)

To secure the routing table maintenance such that malicious nodes can not increase their presence in the routing table, one way is to impose strong constraints on the set of nodeIds that can fill each slot in a routing table. This constraint can be verified and it is independent of network proximity information, which may be manipulated by attackers.

The proposed solution uses two routing tables: one that exploits network proximity information for efficient routing (as in Pastry and Tapestry), and one that constrains routing table entries (as in Chord). In normal operation, the first routing table is used to forward messages to achieve good performance. The second one is used only when the efficient routing technique fails, i.e., the routing failure test failed. In our implementation, we use both tables. In the locality-aware routing table of a node with identifier i , the (r, c) slot can contain any node with nodeId that shares the first r digits with i and has the value c in the $r + 1^{\text{st}}$ digit. In the constrained routing table, the entry is further constrained to point to the closest nodeId to a point p defined as follows: It shares the first ℓ digits with i , it has the value d in the $\ell + 1^{\text{st}}$ digit, and it has the same remaining digits as i .

2.3.4 Secure Routing

After a node has created both PNS routing table and CRT, secure routing works as follows: Given a message for a point X , the message is first routed using the more efficient PNS table. Upon receiving the response, it performs the routing failure test. If it fails, redundant routing is performed using CRT to search for the target replica set.

3 Implementation

This section depicts our implementation as well as optimizations of Constrained Routing Table (CRT). We assume that the reader is familiar with Pastry joining and routing table maintenance protocol [5].

3.1 Bootstrapping

When a new node joins the network, it first contacts some fixed number of bootstrap nodes. All of these bootstrap nodes forward the join message toward the destination. As in the normal join protocol, the routing table members along the path are collected and sent to the joining node. After getting the responses, the joining node constructs its PNS routing table by choosing the low network delay candidate for each entry, and its leaf set by picking the nodes that are closest to its `nodeId`.

After the PNS table is finished constructing, the node considers itself as “active,” i.e., it can notify the application on top that it has successfully joined the network, although the CRT is not yet ready. It then starts to construct its CRT. The construction is done by repeatedly routing to the target `nodeId`: First, each node sends a join message to each of the entry in the CRT with the secure routing mechanism described in Section 2.3.1, using the node’s current leaf set as the distinct starting points. These redundant routing messages are routed using the CRT entries only. The responding node is then added to the CRT, provided that it can pass the routing failure test. If it fails the test, the CRT entry would not be filled, and the process would be repeated at a later time. Note that we never put any node that failed the test into CRT, so as to ensure the quality of the CRT.

The bootstrapping process, if implemented directly in this way, would incur a relatively high overhead. We implement the following optimization to reduce the overhead, as suggested by Castro et al. Observing that nodes close in `nodeId` space are likely to have common CRT entries, a new node first obtains the CRTs from each of its leaf set members. It resorts to redundant routing only if the nodes it obtains cannot pass the `nodeId` density check. Shortly after a new node joins the network, it notifies each node in its CRT of its presence. This provides an opportunity for those nodes to learn the presence of this node and put it in their CRTs. We find experimentally that this dramatically reduces both the delay and the communication overhead to build the CRT.

3.2 Maintenance

CRT maintenance is necessary to ensure the correctness of the CRT under churn. One simple way is to do redundant routing periodically for each CRT entry. The period can be adjusted dynamically according to a local estimation of churn. However, the cost of periodic redundant routing for each CRT entry could be very high.

To this end, we incorporate another optimization. For each CRT entry, we also record the nodeId density of its leaf set. At each maintenance event, each node attempts to update all the empty slots and slots that have a low recorded density (as defined by the routing failure test) in its CRT. First, a request is sent to obtain the latest leaf set of the key using the current CRT. If the response set satisfies the density check (which is likely, since we are using the CRT), the closest node is inserted into the CRT and the density is updated. Otherwise, we resort to redundant routing to update this entry. For each row, if all the entries passed the density check, a random entry is picked to perform this procedure.

3.3 Optimizations for Redundant Routing

Here we describe further the optimizations we have implemented to reduce the overhead of redundant routing. Recall that secure routing first uses PNS table, and if the responding node fails the routing failure test, redundant routing on CRT is used instead. However, falling back to redundant routing on CRT is expensive. The expected number of messages is $O(\ell^2)$, where ℓ is the number of redundant paths used.

3.3.1 Reducing Communication Complexity

Let us break down the cost of redundant routing, as described in Section 2.3.1. Step (1) and (2) requires roughly $\ell \log N + \ell$ messages for a network with size N . Step (3) potentially takes $O(\ell^2)$ messages, since each node receiving a message needs to forward the message to all other members in its replica set who are not in the message. Step (4) involves $O(\ell)$ additional messages. The total number of messages is therefore $O(\ell^2)$, due to Step (3).

Our goal is to reduce the worst-case communication complexity. In our implementation, we only modify Step (3) of the original proposal. When a node received a union set that does not match its replica set, instead of requiring that node to forward the message to all remaining members in its replica set, that node only needs to reply to the originator the missing members in its replica set. The responsibility of contacting those missing members and getting their opinion is left to the originator. In this way, even in the worst case, each of those missing member will only be contacted once by the originator. The cost of this step would be reduced to $O(\ell)$.

Despite the fact that the total cost of redundant routing is now reduced to $O(\ell \log N)$, the absolute cost of redundant routing in terms of message exchange is still quite high. The overhead of the first two steps together already is $\ell \log N + \ell$. For $\ell = 16$, this is 64 messages. Next, we show how we can also reduce this cost.

3.3.2 Progressive Redundant Routing

Redundant routing typically use 16 paths at once. This is usually more than necessary. Assuming that CRTs are constructed correctly, the probability of reaching the correct replica set when starting with ℓ redundant paths can be approximated by

$$1 - (1 - (1 - f)^{1+\log N})^\ell . \tag{1}$$

Figure 1 shows the cumulative probability distribution of successfully reaching the replica set with different values of ℓ and system sizes. Notice that in more than 80% of the time, the correct replica set can

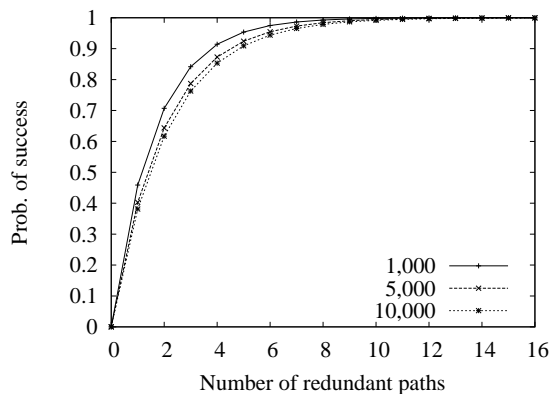


Figure 1: For different values of redundancy, the probability of successfully reaching the replica set.

be reached by using $\ell = 4$ redundant routing paths. We therefore propose *progressive redundant routing*, where redundant routing is performed progressively, starting with a small number of redundant paths. At each iteration, a conservative value of γ is used for routing failure test. If it fails the test, we increase the number of redundant routing paths in each iteration. Since it is statistically more likely that the result is correct with more redundant routing paths, the γ value used in the routing failure test can be relaxed at each iteration to more easily accepting the result.

An additional benefit of progressive redundant routing is that instead of using a fixed γ value, we can tweak the γ value based on our confidence of the result, which depends on the number of redundant paths used. In this way, we can reduce the false negatives (rejecting a correct node) without increasing the false positives (accepting an incorrect node).

4 Attacks and Impacts

In this section, we describe how an adversary, controlling a number of malicious nodes, can mount an Eclipse attack to the network. Basically, malicious nodes can publicize each other to correct nodes, while hiding the identities of correct nodes from other correct nodes. We also describe how our bootstrapping and CRT maintenance protocol can effectively defeat such attacks on CRT.

4.1 Bootstrapping

When a new node joins, malicious nodes can choose to provide only other malicious nodes in their responses. Also, they only forward the join message to other malicious nodes, or pretend to be the node closest to the destination. While a malicious bootstrap node may prevent a new correct node from learning other correct nodes, our requirement of contacting multiple bootstrap nodes ensures that with high probability a new node can join through at least one correct bootstrap node.

4.2 Suppressing Notifications

Malicious nodes can deliberately violate the protocol to hide the existence of correct nodes from each other. For example, they can hold off notifications they are supposed to send when a new node joins the system. Our implementation does not rely on a single node to send all the notifications. For example, all nodes are supposed to send changes in their leaf sets to all of their leaf set members. This prevents a single defected node from hiding such information.

4.3 Manipulating Leaf Sets

Our CRT implementation and optimizations heavily rely on the correctness of routing failure tests. Malicious nodes can manipulate the leaf sets by providing as many malicious nodes as possible while suppressing correct nodes. In our implementation, we periodically use secure routing primitive to discover and update the CRT slots rather than relying on current entries in these slots to provide us with their leafsets.

5 Experiments

We use our CRT implementation to experimentally evaluate the effectiveness of CRT as a defense against Eclipse attacks and its overhead. In the following experiments, we use a static overlay network of 1,000 nodes. For each network, we first construct a network with 16 nodes. Then each addition node joins by first contacting 16 random existing nodes as bootstrap nodes. We assume that the fraction of malicious nodes is $f = 0.2$, all collude with each other to mount an Eclipse attack, as described in Section 4.

5.1 Convergence of CRT entries

The first metric of interest is how quickly the CRT entries converge to the nodes that are closest to their respective keys. Figure 2 shows the fraction of active CRT entries, as well as the *convergence*, i.e., the fraction of active entries that are pointing to a correct node. The result shows that CRT converges very quickly to the correct entries. Within 45 mins of simulation time, more than 95% of the CRT entries are correct.

5.2 Defense against Eclipse attacks

Next, we verify the effectiveness of CRT in defending against Eclipse attacks. Figure 2 also shows that the fraction of malicious entries in the CRT is bounded by $f = 0.2$, same as the fraction of malicious node. Thus, the CRT is perfectly robust against Eclipse attacks as expected.

5.3 Delay Stretch

Figure 3 and 4 compare the delay stretch of the CRT and the PNS-based routing table. Delay stretch is the ratio of overlay route delay relative to the direct IP delay between the source and the destination. The delay stretch penalty for the CRT is roughly the same as the average number of routing hops. This is again

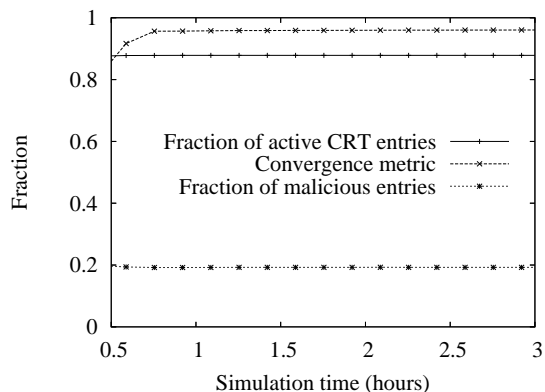


Figure 2: Fraction of active, correct, and malicious entries in CRT over time.

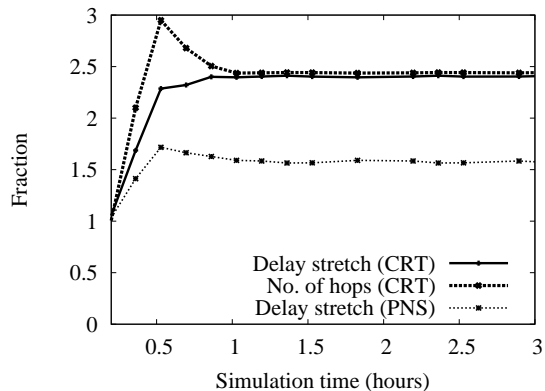


Figure 3: Delay stretch and number of hops for CRT and PNS over time.

expected, since CRT completely ignores proximity. Thus, CRT causes non-local network communication and long delays that increase linearly with the number of hops.

PNS, on the other hand, is an essential overlay optimization to reduce network delay and wide-area network communication. With PNS, the initial hops are short. The total delay is dominated by the delay of the last hop, and so the delay stretch of the entire path is less than two and largely independent of the number of hops taken, as pointed out by Castro et al. [3]. Our results confirm that using the CRT as the primary defense against Eclipse attacks is effective but costly, since it defeats PNS.

5.4 Overhead

Figure 5 shows the per-node message overhead of maintaining the CRT, as well as the total per-node message overhead of maintaining the Pastry overlay, without the optimizations described in Section 3.3. The results show that the additional overhead of maintaining the CRT is marginal. By incorporating the optimizations in Section 3.3, we further reduced the cost of redundant routing by approximately 60%.

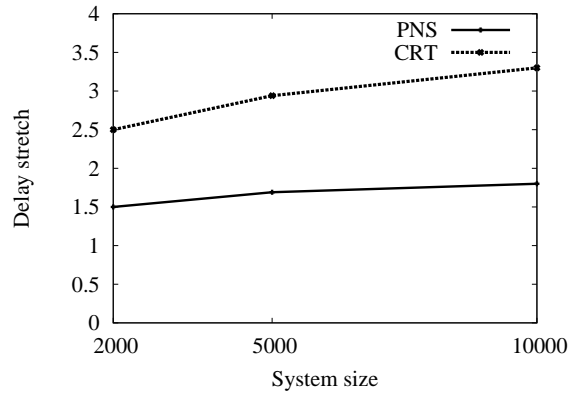


Figure 4: For different system sizes, comparing the delay stretch obtained when using PNS with CRT

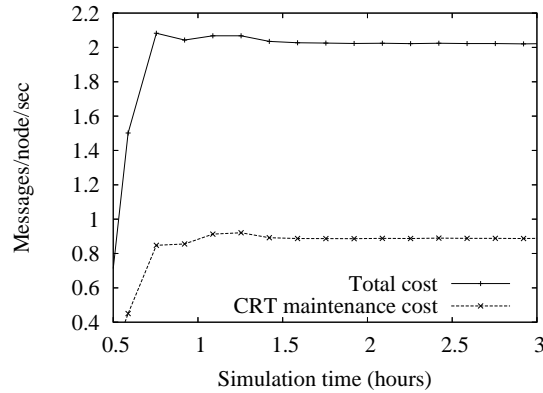


Figure 5: The combined maintenance cost for CRT (redundant routing) and the total cost (including proximity routing table maintenance).

5.5 Delay due to Redundant Routing

We now evaluate the delay introduced due to redundant routing. The delay is defined as the time between a source node detecting normal routing failure and obtaining the replica set.

Recall that with our optimization, redundant routing is performed in multiple iterations. At first, routing is started at 4 different leaf set members. If the replica set obtained in a given iteration does not satisfy the routing failure check, redundant routing is restarted from another 4 leaf set members, and the whole process repeats. After 4 iterations (i.e., 16 redundant paths), however, we assume that redundant routing is successful and accept the replica set without testing.

Figure 6 plots the delay distribution. Observe that more than 70% of time redundant routing is successful within 5 seconds. The discontinuity is due to the different number of iterations redundant routing has gone through. This is confirmed by the fact that in approximately 74% of time redundant routing finishes after the first iteration.

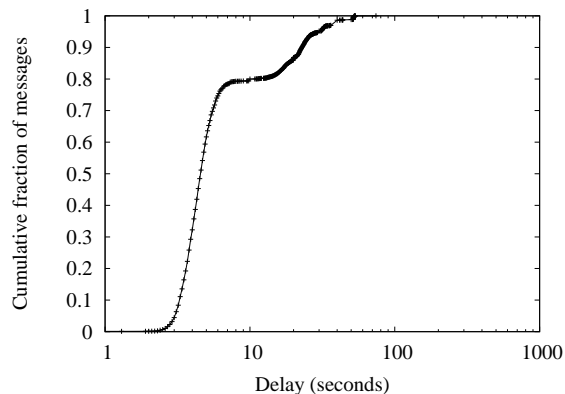


Figure 6: The cumulative distribution of delay introduced by redundant routing.

6 Conclusion

In this paper, we have described our implementation of secure routing primitives, including the optimizations for reducing the message overhead. Our simulation shows that the implementation is correct even under Eclipse attacks, producing high quality constrained routing table. In addition, the overhead is only marginal.

References

- [1] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. OSDI*, Boston, MA, Dec. 2002.
- [2] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, May 2002.
- [3] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Technical Report MSR-TR-2003-52, Microsoft Research, June 2003.
- [4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, San Diego, CA, Aug. 2001.
- [5] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, Heidelberg, Germany, Nov. 2001.
- [6] A. Rowstron, A.-M. Kermarrec, P. Druschel, and M. Castro. Scribe: The design of a large-scale event notification infrastructure. In *Proc. NGC'2001*, London, UK, Nov. 2001.
- [7] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, Mar. 2002.
- [8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, San Diego, CA, Aug. 2001.
- [9] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB-CSD-01-1141, U. C. Berkeley, Apr. 2001.