A shorter version of this paper appears in *Proc. Intl. Conference on Dependable Systems and Networks (DSN-2002)*, Washington DC, USA, IEEE, 2002.

Secure Intrusion-tolerant Replication on the Internet

Christian Cachin Jonathan A. Poritz

IBM Research
Zürich Research Laboratory
Säumerstrasse 4 / Postfach
CH-8803 Rüschlikon, Switzerland
{cca, jap}@zurich.ibm.com

June 25, 2002

Abstract

This paper describes a *Secure INtrusion-Tolerant Replication Architecture*[1] *(SINTRA)* for coordination in asynchronous networks subject to Byzantine faults. SINTRA supplies a number of group communication primitives, such as binary and multi-valued Byzantine agreement, reliable and consistent broadcast, and an atomic broadcast channel. Atomic broadcast immediately provides secure state-machine replication. The protocols are designed for an asynchronous wide-area network, such as the Internet, where messages may be delayed indefinitely, the servers do not have access to a common clock, and up to one third of the servers may fail in potentially malicious ways. Security is achieved through the use of threshold public-key cryptography, in particular through a cryptographic common coin based on the Diffie-Hellman problem that underlies the randomized protocols in SINTRA. The implementation of SINTRA in Java is described and timing measurements are given for a test-bed of servers distributed over three continents. They show that extensive use of public-key cryptography does not impose a large overhead for secure coordination in wide-area networks.

Keywords: Fault Tolerance in Distributed Systems, Security and Cryptography, Byzantine Faults, Asynchronous Networks, State-machine Replication.

1 Introduction

As online services come to play an increasingly important role in today's society, it becomes ever more apparent that the network infrastructure is vulnerable to failures, both caused by unintentional faults and malicious attacks. Recovering from a failure is expensive and often costs much more than the measures which could have prevented it.

Replication is a proven technique for masking component failures. It is well-understood in applications where faults occur randomly and without malicious intent, but its extension to network environments with intentional adversarial behavior is the subject of current research.

This paper describes a *Secure INtrusion-Tolerant Replication Architecture* for asynchronous networks, abbreviated *SINTRA*. It consists of a collection of protocols and their implementation in Java providing secure replication and coordination among a group of servers connected by a wide-area network, such as the Internet. For a group consisting of n servers, it tolerates up to $t < n/3$ servers failing in arbitrary, malicious ways, which is optimal for the given model. The servers are connected only by asynchronous point-to-point communication links and do not have access to synchronized clocks. Thus, *SINTRA* automatically tolerates timing failures as well as attacks that exploit timing. The *SINTRA* group

[1] A previous version of this paper referred to it as *Hydra*.

model is static, which means that failed servers must be recovered by mechanisms outside of *SINTRA*, and the group must be initialized by a trusted process. A detailed discussion of the model is given in [2].

The protocols exploit randomization, which is needed to solve Byzantine agreement in such asynchronous distributed systems [8]. Randomization is provided by a threshold-cryptographic pseudo-random generator, a so-called *coin-tossing* protocol based on the Diffie-Hellman problem. Threshold cryptography is a central concept in *SINTRA* because it allows the group to perform a common cryptographic operation for which the secret key is shared among the servers in such a way that no single server or small coalition of corrupted servers can obtain useful information about it.

SINTRA provides threshold-cryptographic schemes for digital signatures, public-key encryption, and unpredictable pseudo-random number generation (coin-tossing). It contains broadcast primitives for *reliable* and *consistent broadcasts*, which provide agreement on individual messages sent by distinguished senders. These primitives cannot guarantee a total order for a stream of multiple messages delivered by the system, however, which is needed to build fault-tolerant services using the state machine replication paradigm [16]. This is the problem of *atomic broadcast* and requires more expensive protocols based on Byzantine agreement.

SINTRA provides multiple randomized *Byzantine agreement* protocols, for binary and multi-valued agreement, and implements an *atomic broadcast channel* on top of agreement. An atomic broadcast that also maintains a causal order in the presence of Byzantine faults is provided by the *secure causal atomic broadcast channel*.

Experiments have been carried out to measure the performance of a *SINTRA* prototype in a global distributed system with servers in Zürich, California, New York, and Tokyo. They show that the performance of the atomic broadcast protocol with 1024-bit RSA and discrete logarithm public keys is not limited by the cryptography, and lies at a few seconds for providing agreement on the "next" message. Since the prototype is written in Java, the code was not optimized, and the test-bed consisted of cheap, standard machines, it is reasonable to conclude that such protocols are practical today for certain critical applications.

The paper is organized as follows. Section 2 describes the architecture of *SINTRA* and contains brief descriptions of all implemented protocols. The implementation of *SINTRA* in Java and its application program interface (API) are presented in Section 3. Experiments with *SINTRA* in a LAN environment and on a global network are reported in Section 4. Related work is discussed in Section 5, and the paper concludes with Section 6.

2 Architecture and Protocols

SINTRA is designed in a modular way as shown in Figure 1. Modularity greatly simplifies the construction and analysis of the complex protocols needed to tolerate Byzantine faults. A brief account of the protocols implemented by *SINTRA* is given here for completeness; detailed descriptions can be found in companion papers [4, 3, 2].

The system model consists of a static group of n *servers* (also called *parties*), of which up to t may fail. They are connected by reliable asynchronous point-to-point links and have no access to a common clock. Faulty parties can fail in arbitrary, malicious ways and are called *corrupted*, the remaining ones are called *honest*. The protocols of *SINTRA* work under the assumption that $n > 3t$, which is optimal in an asynchronous network with malicious faults.

SINTRA currently needs a trusted dealer to generate the secret keys of all threshold schemes for a particular configuration. The dealer is required only once, when the system is initialized, and the keys must be distributed to all servers in a trusted way. The dealer is needed because it is not known how to generate such keys efficiently in an asynchronous distributed system.

The point-to-point links are authenticated using a message authentication code (MAC) for which one symmetric key for every pair of servers is generated by the dealer. Every server can digitally sign messages using a standard RSA signature scheme, which is needed by some protocols. For simplicity,

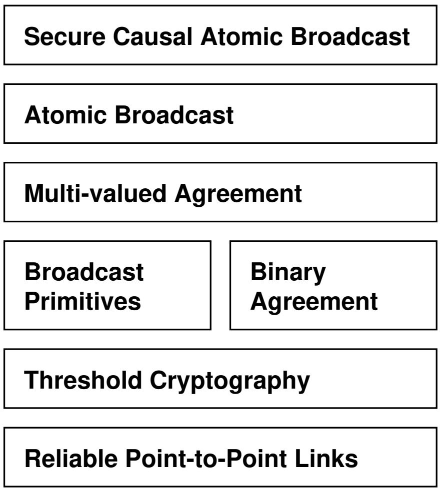

Figure 1: The design of *SINTRA*.

these keys are currently also generated by the dealer and the public keys of all servers are included in the initialization data for a server. However, such keys could in principle be generated efficiently without a trusted dealer.

Every protocol instance is identified by a *protocol identifier*, which must be included in all cryptographic operations of the instance.

2.1 Threshold Cryptography

Threshold cryptography is crucial for several of the protocols in *SINTRA* and forms a core component of the architecture. *SINTRA* uses threshold schemes for *digital signatures*, *coin-tossing*, and *public-key encryption*. They are all non-interactive and robust; they are implemented by a collection of algorithms for generating, verifying, and assembling shares of the cryptographic operation. In this way, they do not require any particular communication pattern and can be easily integrated into asynchronous protocols.

More precisely, *SINTRA* uses the threshold signature scheme of Shoup [17], which produces standard RSA signatures. It may be used by the implementations of consistent broadcast and binary Byzantine agreement. The threshold coin-tossing scheme of Cachin, Kursawe, and Shoup [4], which is based on the Diffie-Hellman problem, plays a crucial role in the randomized binary agreement protocol. For secure causal atomic broadcast, the threshold cryptosystem of Shoup and Gennaro [18] is employed; it is also based on the Diffie-Hellman problem. Coin-tossing and signatures are implemented as (n, k, t) dual-threshold schemes [4]; this means that among the group of n parties, up to t may be corrupted and k (for some $k > t$) shares are needed to generate the unpredictable coin and the digital signature, respectively.

In contrast to the threshold coin-tossing and encryption schemes, standard threshold *signatures* are not essential for the operation of *SINTRA*'s protocols. Since the group is static and a standard digital signature scheme is available whose public keys are known to all servers, it is sometimes more efficient to use a vector of standard signatures instead of a threshold signature. This is called a *multi-signature* and requires no change to the protocols that use threshold signatures. Multi-signatures are particularly suited when computation is more expensive than communication; for example, when a signature need only be verified a small number of times.

All threshold cryptographic schemes can be proved secure only in the so-called random oracle model, which falls short of a proof in the standard model, but nevertheless gives very strong evidence for their security; equally efficient non-interactive threshold schemes in the standard model are currently not known.

2.2 Broadcast Primitives

The two broadcast primitives provided by *SINTRA* are *consistent broadcast* and *reliable broadcast*. Both disseminate a payload message from a distinguished sender to all parties; the identity of the sender is an input parameter to the protocol and must be determined by an external mechanism. Broadcasts are characterized by two local events: *send* occurs only on the sender when the broadcast is started and *deliver* occurs on any receiving server when it accepts a broadcast payload message.

A *reliable broadcast* ensures *agreement* so that all honest parties deliver the same message or nothing at all, in which case the protocol does not terminate. This is implemented by the following protocol due to Bracha and Toueg [1]:

1. the sender sends the payload message to all parties;
2. all parties "echo" the message from the sender to each other;
3. upon receiving $\lceil \frac{n+t+1}{2} \rceil$ such "echo" messages or $t+1$ "ready" messages from others, a party sends a "ready" message to all others;
4. finally, when $2t+1$ "ready" messages have been received, a party accepts the payload message and delivers it.

One feature of this protocol is that it requires no expensive public-key cryptography but only relatively cheap authentication for the point-to-point messages.

In a *consistent broadcast*, the agreement property is relaxed to guarantee only *consistency* of the message among those parties that actually deliver it, whereas some parties may not deliver anything at all. Such a protocol has been used by Reiter [13], who called it "echo broadcast." A broadcast protocol works as follows. The sender sends the payload message to all parties, who generate a share of a threshold signature to bind the payload to the particular broadcast instance and echo it back to the sender (recall that this may also be a multi-signature, which is in fact the protocol proposed by Reiter). Given a quorum of (at least $\lceil \frac{n+t+1}{2} \rceil$) signature shares, the sender obtains a threshold signature on the message and sends it to all parties. A server accepts and delivers the payload message when it receives the corresponding threshold signature.

Consistent broadcast incurs a communication cost that is linear in n, in contrast to reliable broadcast, which has quadratic communication complexity. However, the advantage of consistent broadcast in terms of communication is offset by the more expensive computation needed for the threshold signature generation.

Both broadcast primitives guarantee *termination* only for honest senders. Both also guarantee *authenticity* for honest senders: this means that an accepted payload message is the one that was actually sent by the sender.

2.3 Binary Byzantine Agreement

Binary Byzantine agreement in *SINTRA* uses randomization in order to provide agreement in asynchronous networks, since this is impossible to achieve with deterministic protocols. The protocol ensures that all honest parties agree on a binary value that was proposed by an honest party.

More precisely, a binary agreement protocol starts when a party *proposes* a binary value for the agreement instance, and it terminates when the party *decides* for a binary value. The protocol ensures that all honest parties decide for the same value (*agreement*) and that the decision value has been proposed by at least one honest party (*validity*). Additionally, it guarantees that all parties *terminate* the protocol; for randomized protocols this means that the number of basic steps executed by the protocol is an appropriately bounded random variable.

SINTRA implements the binary agreement protocol of Cachin, Kursawe, and Shoup [4], which uses a cryptographic threshold coin-tossing primitive for randomization. The protocol proceeds in global rounds, each one consisting of three sub-rounds of message exchanges, in which every party executes the following steps:

1. It relays a “pre-vote” message containing its current preference to all others. The pre-vote is a binary value.
2. Based on $n - t$ received pre-votes, it chooses a “main-vote” as follows: if the received pre-votes unanimously contain the same bit, this is made the main-vote; otherwise, the main-vote is set to “abstain.” The main-votes are sent to all others.
3. After collecting $n - t$ main-votes, the party checks if they unanimously indicate the same bit and *decides* for this value if so; otherwise, it releases a share of the threshold coin of the round.
4. The new preference is determined as follows: if a main-vote distinct from “abstain” has been received, it is adopted as the new preference; otherwise, the threshold coin is assembled from a quorum of shares and becomes the new preference.

Furthermore, all votes in the protocol have to be justified by non-interactively verifiable information, such as threshold signature shares and threshold signatures, and only properly justified votes are accepted by the honest parties.

This protocol terminates within an expected constant number of rounds and involves a quadratic expected number of messages, whose length is dominated by the threshold signatures included in them.

The binary agreement protocol is also provided in a form with so-called *external validity* [3]. This means that the initial values are accompanied by a validating “proof” which establishes their validity in a particular context determined by the application. The standard *validity* condition of binary agreement is changed so that an honest party may only decide for a value for which it has corresponding validation data. The validated binary agreement protocol returns this proof together with the decision value.

A binary agreement protocol can be *biased* if an application prefers one decision value over the other one [3]. Such a protocol *always* decides for the preferred value when it detects that an honest party proposed it, even in cases where the other value would have been valid as well. The validated binary agreement protocol above can easily be biased by replacing the output of the threshold coin in the first round by the desired bias.

2.4 Multi-valued Byzantine Agreement

Multi-valued Byzantine agreement provides agreement on values from arbitrary domains. Multi-valued agreement requires a non-trivial extension of binary agreement because the standard validity condition is not useful for multi-valued agreement. For example, one wants to rule out agreement protocols that decide on a value that no party proposed. The solution implemented in *SINTRA* is to use the *external validity* property based on a global predicate with which every party can determine the validity of a proposed value, similar to validated binary agreement. The protocol guarantees that the system may only decide for a value acceptable to honest parties. The remaining properties of multi-valued agreement are the same as in the binary case.

SINTRA uses the multi-valued agreement protocol of Cachin et al. [3], which builds on top of protocols for validated binary Byzantine agreement and consistent broadcast. The basic idea is that every party proposes its initial value as a candidate value for the final result. One party whose proposal satisfies the validation predicate is then selected in a sequence of validated binary agreements and this value becomes the final decision value. More precisely, every party executes the following steps:

1. It sends its own proposed value to all other parties using a *consistent broadcast*. This ensures that the proposal value from any particular party are consistent for all honest receivers. After receiving $n - t$ proposals satisfying the validation predicate, it enters a loop.

2. In each round of the loop, a candidate P_a is chosen in the order given by some permutation Π of $\{1, \ldots, n\}$. Every party carries out the following steps for that P_a:

   (a) if the party has accepted a consistent broadcast containing P_a's proposal, it sends a "yes-vote" message to all parties containing this proposal, and a "no-vote" otherwise;

   (b) it waits for $n-t$ proper vote messages, where "yes-votes" are only counted if a valid proposal from P_a has been received;

   (c) it starts a biased validated *binary* Byzantine agreement protocol and proposes 1 if and only if it has received a valid proposal from P_a, using the threshold signature from the consistent broadcast as a proof for the fact that P_a has made a proposal;

   (d) if the binary agreement decides 1, the party proceeds to step 3, otherwise it repeats step 2.

3. If it has not yet accepted the consistent broadcast by the selected candidate, it obtains the proposal from the validation data returned by the binary agreement.

This protocol takes $O(t)$ executions of the loop and incurs an expected communication cost of $O(tn^2)$ messages.

The order Π in which the parties are considered in the loop can either be fixed as described above, chosen at random from information that is locally available to every party, or chosen at random using the threshold coin-tossing scheme in another round of message exchanges in the first step. The second variation has the advantage of balancing the load among the participants but does not offer more security than a fixed order. The third variation is interesting in combination with an additional protocol step where every party commits to its votes before starting the loop, which reduces the complexity to an expected *constant* number of rounds and $O(n^2)$ messages. The details of this are described in [3]. Only the first two variations are currently implemented in *SINTRA*.

2.5 Atomic Broadcast

An *atomic broadcast* guarantees a total order on messages, so that all honest parties deliver the same sequence of messages. Given an atomic broadcast primitive, a fault-tolerant replicated service can be implemented immediately by distributing all state updates using atomic broadcast.

Atomic broadcast differs from the protocols described so far in that it is a continuous protocol with on-line inputs and outputs, in contrast to the isolated instances of broadcast primitives and agreements. This is the reason for implementing atomic broadcast as a *channel* in *SINTRA*.

A broadcast channel is characterized by local *send* and *deliver* events; a party may execute *send* multiple times and must be prepared to *deliver* as many payload messages as the channel outputs. Atomic broadcast ensures that all honest parties deliver the same sequence of payload messages (*agreement* and *total order*) and that a payload message known to at least f parties is delivered after a bounded delay (*fairness*), for $t + 1 \leq f \leq n - t$.

The basic structure of *SINTRA*'s atomic broadcast protocol resembles the atomic broadcast protocol of Chandra and Toueg [6] for the crash-failure model: the parties proceed in global rounds and agree on a batch of messages to deliver using multi-valued Byzantine agreement.

In each round, every party first signs the next message to *send* together with the current round number, and sends this to all other parties. If no message to send is available locally, a party may also adopt a message that was first signed by another party and sign that. Every party then proposes a batch of $n - f + 1$ properly signed messages for multi-valued agreement (the *batch size* $n - f + 1$ is a configurable parameter). The external validity condition must verify that all messages in a batch come with valid signatures from distinct parties; this implies that at least $n - f - t + 1$ payload messages in a batch have been signed by honest parties. All messages in the agreed-upon batch are then delivered according to a fixed order. Fairness is maintained once a particular message m is known to at least f

honest parties because in every round of the protocol, at least one message is delivered that was initially signed by some honest party who also knows m.

The details of the atomic broadcast protocol can be found in [3]. The protocol requires the dealer to generate a key pair of a digital signature scheme for every party and include the public keys of all parties in the initialization data.

The atomic broadcast channel in *SINTRA* uses the sender's identity and a separate sequence number for each sender for identifying the payload messages. This is in contrast to the abstract protocol description above and in [3], where payloads are identified by their bit-string representation. This change seems unavoidable for any reasonably efficient implementation since one would otherwise have to store a complete history of payloads. Thus, the abstract *integrity* property that every payload message is delivered at most once, no matter which parties sent it, has to be weakened: In *SINTRA*, *integrity* means that a message, consisting of a bit string, is delivered at most once for every time that an honest party has sent that bit string. If an application wants to exploit the fairness property with multiple parties sending the same message as bit string, it must perform this on top of atomic broadcast using an extra layer to filter out the duplicates. This relaxation of the ideal model is an example of how an end-to-end principle has to be violated to make an implementation practical [15].

Because atomic broadcast is an on-line protocol with an a priori unknown number of inputs and outputs, it requires a special mechanism for termination. In *SINTRA*, a party signals that the channel may be closed using a local event *close*. The protocol implementation then sends a termination request message on the channel as if it were a regular payload. When the channel outputs such a termination request, a counter is increased for every party who sent one, and the protocol terminates after the round in which this number reaches $t+1$. Thus, the channel is guaranteed to terminate when all honest parties together *close* it, and the channel is kept open unless at least one honest party *closes* it.

2.6 Secure Causal Atomic Broadcast

Secure causal atomic broadcast augments atomic broadcast with confidentiality for the payload messages until their position in the sequence of delivered payloads is determined. This concept was introduced by Reiter and Birman [14] in order to ensure a causal order among all payloads in the presence of Byzantine faults.

Secure causal atomic broadcast combines an atomic broadcast channel with a robust threshold cryptosystem [14]. In order to *send* a payload message, the sender encrypts it under the global public key of the channel and *sends* the ciphertext on the atomic broadcast channel. Whenever the channel *delivers* a ciphertext, all parties release and exchange a decryption share for the ciphertext using an additional round of interaction. After obtaining a quorum of decryption shares, the cleartext payload message is recovered and *delivered* locally.

The threshold cryptosystem must be secure against adaptive chosen-ciphertext attacks to prevent a malicious party from modifying a ciphertext into anything related to the payload message. Thus, messages sent on this broadcast channel remain confidential until after their position in the sequence is determined, which means that causality is maintained. Secure causal atomic broadcast provides only the weaker form of integrity mentioned above.

SINTRA implements the secure causal atomic broadcast protocol sketched above. It uses the non-interactive threshold cryptosystem of Shoup and Gennaro [18], for which the key shares have to be distributed by the dealer.

2.7 Aggregated Broadcast Primitives

Sometimes an application may want to broadcast multiple messages to the group without the need for totally ordered delivery. Clearly, using atomic broadcast in this case is overkill, but using the reliable broadcast primitive is cumbersome since the application must manage a separate broadcast instance for every possible sender. The *reliable channel* and *consistent channel* abstractions fill this gap by providing virtual channels that aggregate many instances of the corresponding broadcast primitive together.

A reliable channel provides the same interface as atomic broadcast for sending multiple messages. Internally, however, a reliable channel runs n reliable broadcast instances in parallel, one for every party, and allocates a new one for every instance that terminates. A request to *send* a message is handled by the current instance of the sender. A message received by the broadcast instance is multiplexed back onto the channel to be *delivered*, and a new broadcast instance is allocated for the sender with its sequence number increased by one. A reliable channel guarantees *agreement* but no ordering.

A consistent channel works analogously, but provides only *consistency* for the delivered messages.

Termination is handled in a similar way as for atomic broadcast: in order to *close* the channel, a party sends a special termination request message as its last message. Every party counts the number of such requests that it receives; when this number reaches $t+1$, it *aborts* the broadcasts that are still active and terminates.

Reliable and consistent channel are examples of virtual protocols that do not exchange any messages of their own over the network.

These channel protocols guarantee weaker properties than atomic broadcast, which may be sufficient for certain applications. They offer a cheap alternative to atomic broadcast, in particular when combined with external means to provide agreement about which messages have actually been delivered. For example, Malkhi, Merritt, and Rodeh [11] propose an external "stability mechanism" with this effect; their WAN broadcast protocol corresponds to *SINTRA*'s consistent channel combined with such a stability mechanism.

3 Java Implementation

The Java implementation of *SINTRA* organizes the protocols presented above in a class hierarchy, in which all protocol classes are extensions of an abstract class `Protocol` that defines a protocol identifier `String pid`.

```
class Protocol {
  Protocol(String pid);
}
```

Every protocol running in *SINTRA* is represented by an instance of `Protocol` and uniquely identified by its `pid`.

The three abstract classes `Broadcast`, `Agreement`, and `Channel` represent the major protocol types in *SINTRA*. Their APIs differ according to services provided by these protocols. The protocols themselves are extensions of these base classes. The following protocols are implemented:

`Broadcast` — `ReliableBroadcast` and `ConsistentBroadcast` provide reliable and consistent broadcast, respectively (Section 2.2);

`Agreement` — `BinaryAgreement` provides binary Byzantine agreement (Section 2.3), `ValidatedAgreement` provides validated binary Byzantine agreement (Section 2.3), and `ArrayAgreement` provides multi-valued agreement (Section 2.4);

`Channel` — `AtomicChannel` provides an atomic broadcast channel (Section 2.5), `SecureAtomicChannel` provides a secure causal atomic broadcast channel (Section 2.6), and `ReliableChannel` and `ConsistentChannel` provide reliable and consistent broadcast channels, respectively (Section 2.7).

The structure of these classes together with their most important methods is shown in Figure 2.

The *SINTRA* implementation in Java is multi-threaded and uses a separate thread per active protocol instance. These threads communicate by sending asynchronous events, which are listed in connection

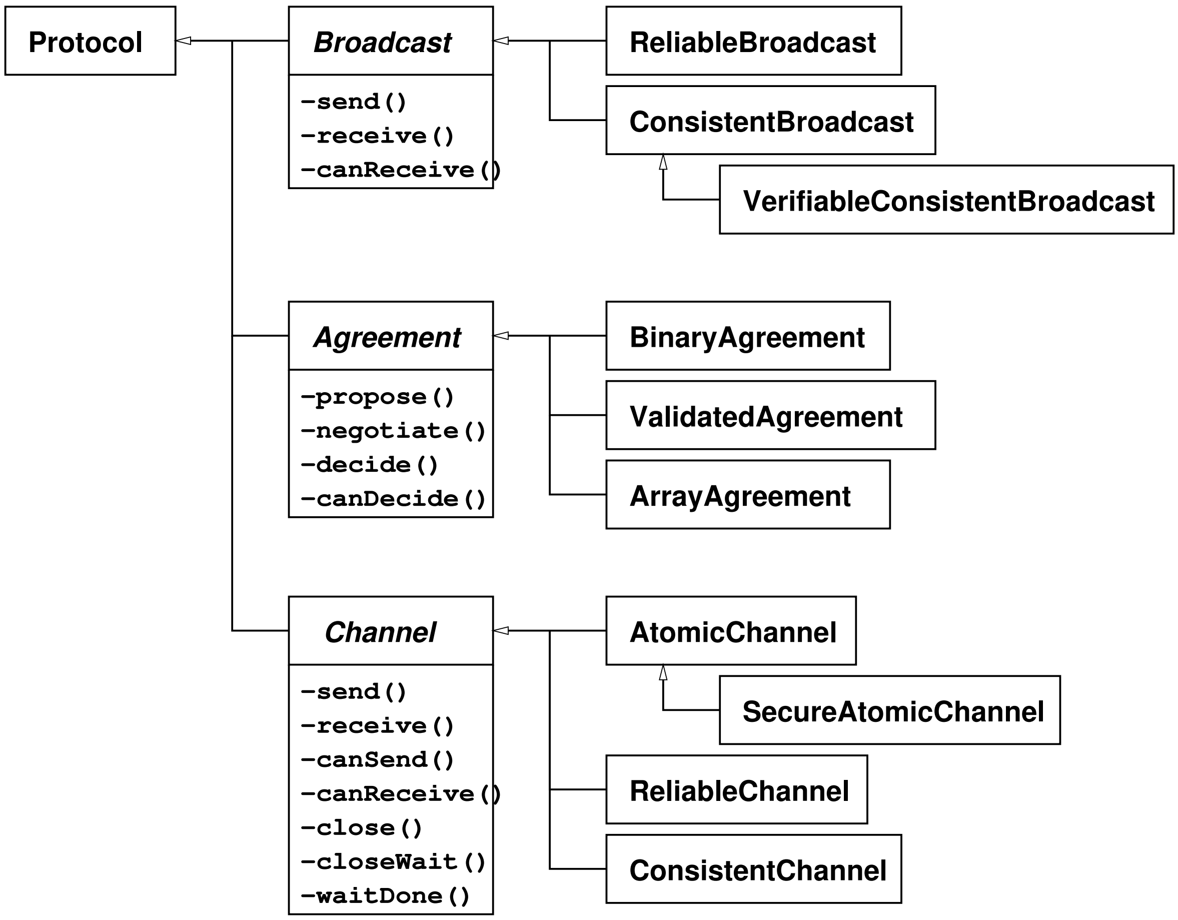

Figure 2: The static class structure of *SINTRA*.

with the protocol APIs below. The reliable point-to-point links are currently implemented by TCP streams for simplicity and are therefore subject to a denial-of-service attack by sending forged TCP acknowledgements. It is planned to replace TCP by *SINTRA*'s own sliding-window implementation, which will provide authenticated acknowledgments.

Link authentication is provided over TCP using HMAC with SHA1 and a 128-bit shared key for every pair of servers. SHA1, which generates 160-bit outputs, is also used as the hash function in the threshold signature and standard signature schemes, and in the threshold coin-tossing protocol.

SINTRA uses a configuration file that contains all important parameters, such as the identities of all parties, the system parameters n and t, the cryptographic key sizes etc. A party is identified by an Internet address of the form `hostname:port`, which denotes the socket endpoint through which it connects to the others. Internally, the parties are named by an integer index between 0 and $n - 1$.

SINTRA has been implemented in the Java 1.2 platform.

3.1 Threshold Cryptography

SINTRA contains threshold implementations of three cryptographic primitives: digital signatures, pseudo-random number generation or "coin tossing," and public-key encryption. The schemes mentioned in Section 2.1 are implemented by the classes `ThresholdSignature`, `ThresholdCoin`, and `ThresholdCipher`. As an example for all three of them, only the coin-tossing scheme is described here. `ThresholdSignature` is an abstract class that is implemented by `RSAThresholdSignature`, providing "proper" RSA threshold signatures [17], and by `MultiSignature`, providing multi-signatures with a vector of digital signatures from *SINTRA*'s `Signature` class.

The threshold cryptography classes are roughly modeled after the implementations of signature and encryption schemes in the Java cryptography library (JCE 1.2.1), in particular after the packages `java.security` and `javax.crypto`. For simplicity, however, *SINTRA* makes no separation between the interface and dynamically configurable providers, and keys are represented directly as `BigInteger` objects.

`ThresholdCoin` has the following interface:

```
class ThresholdCoin{
  ThresholdCoin(int keySize, int modSize, int n, int k, int t);
  void initRelease(BigInteger privateKey, BigInteger[] globalVerifyKey,
                   BigInteger localVerifyKey);
  void initVerifyShare(BigInteger[] globalVerifyKey,
                       BigInteger localVerifyKey);
  void initAssemble(BigInteger[] globalVerifyKey);
  void update(byte[] b);
  byte[] release();
  boolean verifyShare(byte[] share);
  byte[] assemble(byte[][] shares, int len);
}
```

The constructor generates a new instance of the (n, k, t) dual-threshold coin with the key size and modulus size in bits. The default key size (i.e., the logarithm of the group order) is 160 bits with a modulus of 1024 bits, but all key sizes can be chosen dynamically.

A threshold coin instance operates in three modes: it may *release* a share, *verify* a share, or *assemble* k shares to the random value. The mode must be specified by calling the corresponding `init` method first. The data upon which the operation is performed is then passed using one or more calls to `update`. For the threshold coin, the data is the "name" of the coin, i.e., an arbitrary bit string on which the pseudo-random function is evaluated. Finally,

- either a new share of the named coin is returned by calling `release()`, generated from the private key passed to `initRelease`
- or the validity of a putative share contained in a byte array `share`, with respect to the keys specified with `initVerifyShare`, is verified using `verifyShare(share)`,
- or an array of k valid shares is assembled and the resulting pseudo-random value of `len` bytes is returned by calling `assemble`.

Afterwards, the threshold coin instance may be initialized for the next operation.

3.2 Broadcast

The *reliable broadcast* and *consistent broadcast* primitives (Section 2.2) both implement the following `Broadcast` interface:

```
class Broadcast extends Protocol {
  Broadcast(String basepid, int sender);
  int getSender();
  void send(byte[] message);
  byte[] receive();
  boolean canReceive();
  void abort();
}
```

In the constructor, the `pid` of the protocol instance is not specified directly; instead, `Broadcast` takes a `basepid` argument together with the index of the sender. The `pid` is then set to `basepid + "." + sender`. The service of a broadcast instance is accessed through a non-blocking call to `send(message)` with a byte array argument `message`, which may only be executed by the `sender`, and by a blocking call to `receive()`, which returns the delivered payload message as a byte array. Client code that cannot afford to block until the payload message is ready may use `canReceive()` to determine this first. Note that the sender must execute `send` exactly once, and that `receive` returns at most once. Finally, calling `abort()` provides a way to terminate a broadcast instance immediately. The local instance of the protocol is cleaned up, but the state of other parties engaged in the protocol is unspecified.

Inside *SINTRA*, these calls are mapped onto the following local events:

SEND — Carries an input to the `Broadcast` instance, containing a broadcast request with a payload message from a higher-level protocol or application.

DELIVER — Represents the output from the `Broadcast` instance to a higher-level protocol or application and contains the payload message.

ABORT — Forces the protocol instance to terminate immediately.

The classes `ReliableBroadcast` and `ConsistentBroadcast` extend `Broadcast` by implementing the corresponding broadcast protocols, but do not add anything to the API. This concludes the description of the reliable broadcast and consistent broadcast APIs in *SINTRA*.

Consistent broadcast is an example of a *verifiable* broadcast protocol [3], which means that a party who has delivered the payload message can produce a single protocol message that allows any other party to deliver the payload and terminate the broadcast as well. Such a message is called the *closing message* of a broadcast. *SINTRA* provides verifiable consistent broadcast in a class `VerifiableConsistentBroadcast` that extends `ConsistentBroadcast`. This is a virtual protocol on top of consistent broadcast that requires no additional communication with other parties and uses the following interface:

```
class VerifiableConsistentBroadcast extends ConsistentBroadcast {
  byte[] getClosing();
  void deliverClosing(byte[] v);
  static byte[] getPayloadFromClosing(byte[] v);
  static boolean isValidClosing(String pid, byte[] v);
}
```

The first method `getClosing()` returns a byte array containing the closing message from an already terminated instance, which can then be encapsulated in a message of another protocol and sent to another group member. The other party may execute `deliverClosing(v)` with a byte array `v` obtained in this way. If `v` is a valid closing message for the instance, the party will deliver the payload without waiting for any further network messages and then terminate the broadcast instance.

In the consistent broadcast protocol of Section 2.2, the closing message consists of the payload message and the threshold signature that binds the payload to the instance. This mechanism is used, for example, in the multi-valued agreement protocol of Section 2.4 to prove that a candidate P_a has made a valid proposal.

The static method `getPayloadFromClosing(v)` extracts the payload of a consistent broadcast from a closing message `v` and `isValidClosing(pid, v)` determines if `v` represents a valid closing message for the consistent broadcast instance `pid`.

3.3 Agreement

The `Agreement` interface groups together the common functions of the three protocols for *binary*, *validated*, and *multi-valued* Byzantine agreement in *SINTRA* (Sections 2.3 and 2.4). Since the data types used by the three forms differ, the abstract class `Agreement` is defined in terms of a `Negotiable` interface:

```
class Agreement extends Protocol {
  void propose(Negotiable value);
  Negotiable negotiate(Negotiable value);
  Negotiable decide();
  boolean canDecide();
  void abort();
}
```

A `Negotiable` object stands for the subject of an agreement protocol, which may be a binary value, a binary value with a validating "proof," or an arbitrary bit string. `Negotiable` objects are only used internally in *SINTRA* and are not needed for client programs.

In all variations of agreement, a party executes `propose(value)` to start the agreement instance and *propose* a `value`, and it obtains the decided value from a call to `decide()`, which blocks until the protocol *decides*. The method `negotiate(value)` is shorthand for proposing `value` and then returning the decided value. Client code may use `canDecide()` to determine if a subsequent call to `decide()` will not block. Every party must propose some value once and decides exactly once. Calling `abort()` provides a way to terminate an agreement instance immediately. The local instance of the protocol is cleaned up, but the state of other parties engaged in the protocol is unspecified.

These calls are mapped onto the following local events:

`PROPOSE` — Starts the `Agreement` instance with a value proposed by an application or a higher-level protocol.

`DECIDE` — Carries the decision value of the `Agreement` instance and signals its termination.

`ABORT` — Forces the protocol instance to terminate immediately.

The three incarnations of `Agreement` found in *SINTRA* are described next.

Binary Agreement The binary agreement protocol of Section 2.3 is provided by the following class:

```
class BinaryAgreement extends Agreement {
  void propose(boolean value);
  boolean negotiate(boolean value);
  boolean decide();
}
```

Validated Agreement The validated (binary) agreement protocol mentioned at the end of Section 2.3 is provided by the following class:

```
class ValidatedAgreement extends Agreement {
  ValidatedAgreement(String pid, boolean bias);
  void propose(boolean value, byte[] proof,
               BinaryValidator validator);
  boolean negotiate(boolean value, byte[] proof);
  boolean decide();
  byte[] getProof();
}
```

As for `BinaryAgreement`, the subject of the negotiation is a `boolean`; in addition, the caller has to supply a byte array `proof` and a validator object `validator` in the call to `propose`. The validator is of the abstract type `BinaryValidator`, which ensures that it contains a handle to a validation method of the form

```
boolean isValid(boolean value, byte[] proof)
```

The `ValidatedAgreement` implementation makes repeated up-calls to this. If necessary, the proof that establishes the validity of the decided value can be obtained by calling `getProof()`.

The two-argument constructor provides a way to create a *biased* validated agreement instance that is biased to `bias`.

Array Agreement Multi-valued agreement as described in Section 2.4 is called *array agreement* in *SINTRA*. It is implemented by the class `ArrayAgreement`. The subject of its negotiation is an arbitrary byte array:

```
class ArrayAgreement extends Agreement {
  void propose(byte[] value,
               ArrayValidator validator);
  byte[] negotiate(byte[] value,
           ArrayValidator validator);
  byte[] decide();
}
```

The method `propose(value, validator)` starts the agreement protocol with the proposed byte array `value` and an object `validator` of the abstract type `ArrayValidator`. The decision value is returned as a byte array either by `decide()` or by `negotiate`.

`ArrayValidator` is an interface ensuring that the `validator` object contains a handle to a validation method of the form

```
boolean isValid(byte[] value)
```

Unlike in the binary case, a separation of the value from the corresponding "proof" is not necessary here.

3.4 Channel

The implementations of the *atomic broadcast*, *secure causal atomic broadcast*, *reliable channel*, and *consistent channel* protocols use the abstract `Channel` interface:

```
class Channel extends Protocol {
  void send(byte[] message);
  byte[] receive();
  void close();
  void closeWait();
  void waitDone();
  boolean canSend();
  boolean canReceive();
  boolean isClosed();
  void abort();
}
```

An application may *send* a byte array `message` on the `Channel` by calling `send(message)`, which may block if the `Channel` is congested and all buffers are full. Applications that do not want to be blocked may call `canSend()` first to find out if a subsequent `send` will not block. Every party may *send* any number of messages.

The payload messages output by the channel are *received* by calling `receive()`, which returns a byte array. Every party must be prepared to call `receive()` for an arbitrary number of payloads until the channel closes. If the outputs are not removed like this, the channel will stall and eventually block the parties who are sending. Calling `canReceive()` allows an application to determine that a subsequent `receive()` will not block.

When the application has determined that it is ready to *close* the channel, it may call `close()`. It may then continue to *receive* some messages until `isClosed()` is true, or call `waitDone()`, which will block until the channel has terminated. Instead of calling `close()` and `waitDone()` separately, the application may call also `closeWait()`, which signals the termination and returns only after the channel has been closed.

Finally, calling `abort()` provides a way to terminate an agreement instance immediately. The local instance of the protocol is cleaned up, but the state of other parties engaged in the protocol is unspecified.

These calls are mapped onto the following local events internally:

SEND — Represents an input broadcast request to the `Channel` instance with a payload message from an application or a higher-level protocol.

DELIVER — Represents the output from the `Channel` instance to a higher-level protocol or application.

CLOSE — Signals that the `Channel` instance may be closed.

CLOSEDONE — Signals that the `Channel` instance has been closed and will not generate more local events.

ABORT — Forces the protocol instance to terminate immediately.

`AtomicChannel`, `ReliableChannel`, and `ConsistentChannel` extend the class `Channel` by filling in the corresponding protocol implementations, but do not add any additional functions.

The *secure causal atomic broadcast* implementation is the only channel protocol that extends the `Channel` interface. It is realized on top of an *atomic broadcast channel* by the `SecureAtomicChannel` class:

```
class SecureAtomicChannel
  extends AtomicChannel {
  static byte[] encrypt(String pid, BigInteger
    channelPublicKey, byte[] message);
  sendCiphertext(byte[] ciphertext);
  byte[] receiveCiphertext();
  boolean canReceiveCiphertext();
}
```

The first two methods allow an entity who is not a member of the *SINTRA* group to send a byte array `message` to the group on the secure atomic channel; it only needs to know the global `channelPublicKey` associated with the channel instance to `encrypt` it. The resulting byte array `ciphertext` must be sent to sufficiently many group members by another mechanism. These parties should then broadcast it to the group as a whole by calling `sendCiphertext(ciphertext)`, without seeing the cleartext message.

If needed, an application can access the point in time when the next output from the channel is determined (but not yet decrypted) by calling `receiveCiphertext()`, which returns the encryption of the next payload that will be *received.* By calling `canReceiveCiphertext()` first, an application can ensure that this call will not block. Calling `receiveCiphertext()` is optional and superseded by the subsequent call to `receive()`, which returns the cleartext.

4 Experimental Results

Several experiments have been conducted with *SINTRA* in local- and wide-area networks. We describe their results in detail for atomic broadcast since it is the most important protocol in *SINTRA*.

The local-area network setup consists of four servers with different operating systems and Java virtual machines, connected by the 100 Mbit/s switched Ethernet at the IBM Zurich Research Laboratory (in the CPU column, P3 means a Pentium III and 604 a PowerPC 604, the MHz column gives the clock rate, and the 'exp' column the time to perform a 1024-bit modular exponentiation in milliseconds):

	OS	CPU	MHz	Java	exp
P0	Linux 2.2.x	P3	933	1.3.1 (Sun)	93
P1	Linux 2.2.x	P3	800	1.3.0 (IBM)	70
P2	AIX 4.3	604	332	1.2.2 (IBM)	105
P3	Win2k	P3	730	1.2.2 (IBM)	132

(P0 is a NetVista A40p, P1 a Thinkpad T21, P2 an RS/6000 43P type 7043, and P3 an IBM PC300 PL3.)

The Internet setup consists of four servers on three different continents, located at IBM Research labs in Zürich, Tokyo, New York, and California, connected by the IBM intranet. All systems are standard, Intel-based PCs running Linux (2.2.x kernels):

	Location	CPU	MHz	Java	exp
P0	Zürich	P3	933	1.3.1 (Sun)	93
P1	Tokyo	P3	997	1.3.0 (IBM)	55
P2	New York	P3	548	1.3.0 (IBM)	101
P3	California	P Pro	200	1.3.1 (Sun)	427

The average network latency between these servers is shown in Figure 3; the packet round-trip times range from about 100 to 400ms between most pairs of hosts. The average latency was measured several times during the tests and its variation is quite large, often 10% or more.

The Java interpreters with version 1.3.1 are running the "server" virtual machine (JVM) and the others have just-in-time compilation enabled. The large differences in the performance of modular exponentiation can be attributed to better just-in-time compilation on the IBM JVMs.

The public keys for the discrete logarithm-based threshold schemes (coin-tossing and encryption) use a 1024-bit prime p such that $p - 1$ has a 160-bit prime factor. The digital signature scheme, which is used for the multi-signature implementation of threshold signatures and for the atomic broadcast protocol, is implemented using RSA; its public keys and those of the standard threshold signature scheme (which is also based on RSA [17]) use 1024-bit moduli. Threshold encryption needs also a block cipher for bulk encryption, for which MARS is used with 128-bit keys.

Each one of the above installations alone represents the minimal system configuration of *SINTRA* with $n = 4$ and $t = 1$; taken together they form a hybrid local- and wide-area network configuration with $n = 7$ and $t = 2$ (the machine P0/Linux in Zürich is part of both setups). The batch size of the atomic broadcast channel is set to $t+1$, and the agreement order is determined randomly from local information (cf. Π in Section 2.4). Threshold signatures are implemented as multi-signatures (cf. Section 2.1) if nothing else is mentioned. The test program opens a channel to broadcast messages and has one or more servers send short payload messages (< 32 bytes) to the group at maximum capacity. Then the elapsed time between successive delivery of two messages is measured on a recipient.

4.1 Protocol Behavior

Figure 4 shows the details of an *atomic broadcast* test run (using `AtomicChannel`) on the LAN, where three servers with different operating systems (P0/Linux, P2/AIX, and P3/Win2k) send a total of 1000 messages concurrently. The measurement occurs on the Linux server (P0). The striking feature is the presence of two bands of data points at 0s and at 0.5–1s.

This separation results from the batching in the atomic broadcast protocol. Recall that the protocol proceeds in rounds and delivers a batch of two messages in every round. Since the second message is output immediately after the first one, it shows up at 0s after the previous one in the graph. Moreover, the order of going through the batch is fixed and corresponds to the index of the sender, which explains why messages from P0/Linux are always output first in the round and those at 0s are mostly from P2/AIX and P3/Win2k.

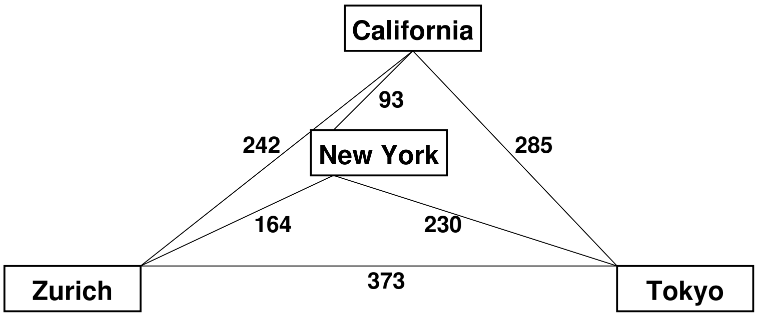

Figure 3: The experimental setup on the Internet, with average round-trip times in milliseconds.

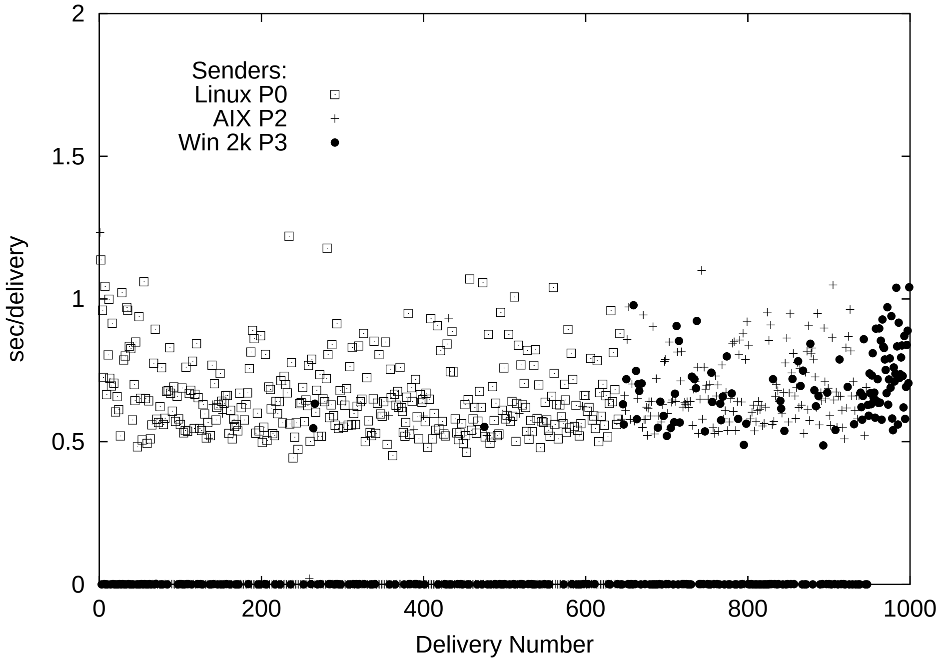

Figure 4: Delivery time per message for `AtomicChannel` on a LAN.

The second visible feature is that the messages sent by the three servers are not delivered in a uniform way: all messages from P0/Linux are delivered first together with some from P2/AIX and P3/Win2k, then P2/AIX and P3/Win2k continue together, and the last 50 messages are only from P3/Win2k. This is because P2/AIX and P3/Win2k are apparently slower than P0/Linux, and P3/Win2k is slower than P2/AIX, and because the protocol considers the messages in the order in which they arrive in the current round. The sender has to digitally sign the message together with the round number, which is computationally expensive. Since the LAN is very fast, only some of the proposed messages from slower machines make it into a batch assembled by any of the faster ones, as long as a faster machine is sending.

On the Internet, a similar pattern results as shown in Figure 5, but two bands are now visible at 2–2.5s and at 3–3.5s. The experiment is the same as on the LAN, with three senders in Zürich, Tokyo, and New York, and the measurement taken in Zürich. The increased network latency multiplies the average delivery time by a factor of about four compared to the LAN.

The distinction between the upper two bands is caused by the randomized order in which the candidates in multi-valued agreement are considered: the protocol waits only for $n - t$ proposals before starting the sequence of binary agreements, and if a slower remote server happens to be the first candidate under consideration, it may be that none of the faster servers, which start binary agreement together, has received its proposal. Thus, the candidate is rejected and a second binary agreement becomes necessary, which explains why roughly 1/4 of the data points (among those above 0s) are grouped together at 3–3.5s. The difference of about 1s corresponds to the time to run one binary agreement instance.

In contrast to the LAN setup, the delivery order is not determined by the sender's speed, but by its connectivity. The slowest sending machine (New York) comes through first (in deliveries 0–500) and apparently makes up for its lack of speed by being closer to enough fast servers. Most messages sent by the Zürich server arrive before those from Tokyo. The last ca. 300 message deliveries are only from the Tokyo server. This is because the Tokyo server is the most difficult to reach from the others, although it has the fastest processor and fast modular arithmetic. The delivery time also increases and varies more towards the end of the run.

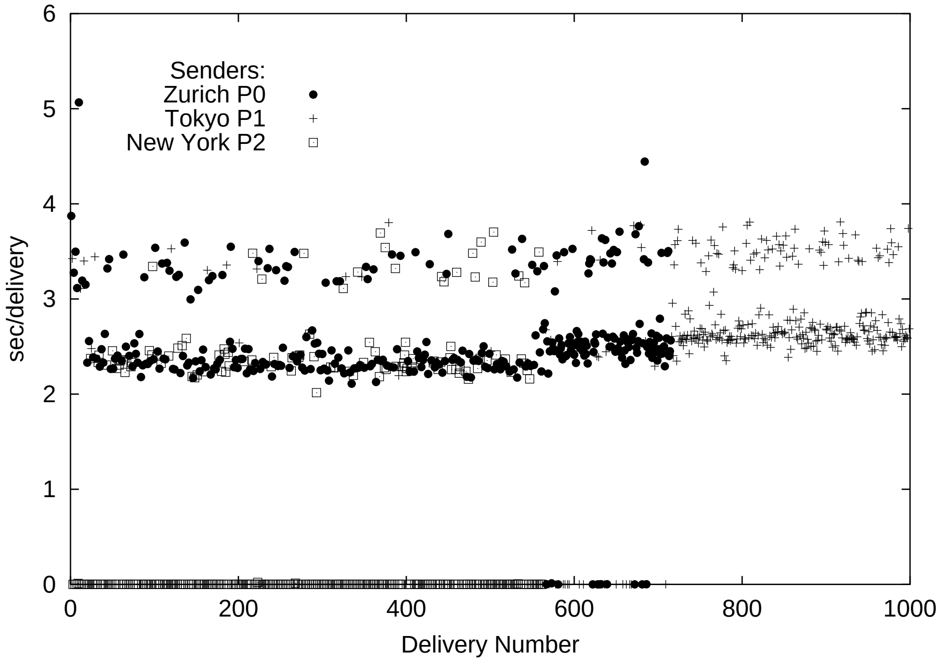

Figure 5: Delivery time per message for `AtomicChannel` on the Internet.

4.2 Performance Measurements

To measure the latency of the protocols implementing the `Channel` interface, the tests reported here have been run with only one server sending 500 messages. The rest of the setup is the same as in the previous section. With one sender, the atomic broadcast protocol runs one multi-valued agreement for every delivered message.

Table 1 shows the average times between successive deliveries on a single server for *atomic channel*, *secure causal atomic channel*, *reliable channel*, and *consistent channel*, with the LAN setup, the Internet setup, and all machines together are as follows. The sender is always P0/Linux in Zürich.

In most settings, reliable channel is the fastest protocol. Apparently, the digital signature operations for consistent broadcast cost more than the larger number of messages for reliable broadcast. A message delivery on the atomic broadcast channel takes four to six times longer than on the consistent or reliable channels. The additional threshold decryption round of the secure causal atomic broadcast adds another 500ms–1s to the latency on the atomic broadcast channel.

There is a surprisingly small performance difference between the Internet setup with four machines and the setup with all seven machines. Most protocols are actually *faster* with more machines. Although it is difficult to find a single explanation for this behavior given the differences in performance of the involved servers, a key factor may be better load balancing in the combined setup: assuming all machines on the LAN are ready to proceed, only one more answer from a remote server is needed for the combined setting, in contrast to two answers from remote servers for the Internet setting.

Setup	atomic	secure	reliable	consistent
LAN	0.69	1.07	0.13	0.11
Internet	2.95	3.61	0.72	0.83
LAN+I'net	2.74	3.79	0.60	0.64

Table 1: Average delivery times (s) for *atomic channel*, *secure causal atomic channel*, *reliable channel*, and *consistent channel*.

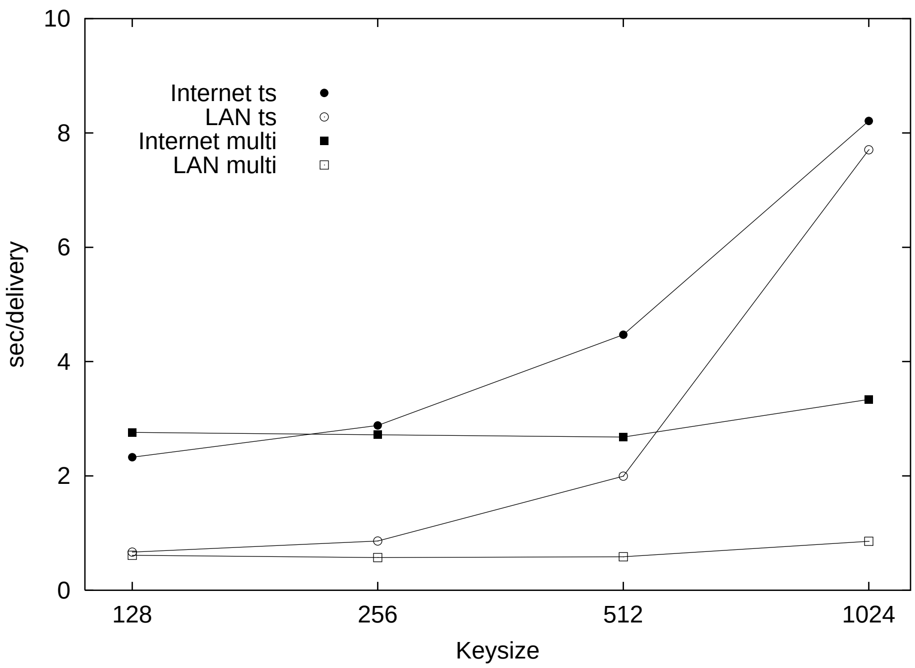

Figure 6: Average delivery time versus size of the public keys with standard threshold-signatures (ts) and multi-signatures (multi).

Figure 6 shows the influence of reducing the public-key size from 1024 bits to artificially small values. The figure shows measurements for the LAN and Internet settings and with RSA-based threshold-signatures and multi-signatures (i.e., threshold signatures implemented by a vector of ordinary digital signatures). The multi-signature implementation benefits also from fast modular exponentiation using Chinese remaindering [12] in the standard RSA signature scheme. Since the complexity of the expensive public-key operations is quadratic in the key size (modular multiplication) or even cubic (full-size modular exponentiation), one might expect that delivery time would increase by at least a factor of four in every step, if the cryptography were to dominate the workload. However, with *multi-signatures* the key length has no significant influence on the delivery time, not even on the LAN. The increased message length makes the multi-signature implementation slightly slower than threshold signatures in the Internet setup with very small keys.

With standard *threshold signatures*, the influence of the key size is visible, but only above 256 bits. In the LAN setting, the step from 512-bit keys to 1024-bit keys causes an increase in delivery time by a factor of almost four. In the Internet setting, delivery time always increases by less than a factor of two when the key size is doubled.

It is apparent that protocol overhead and network delays, but not cryptographic operations, account for most of the time taken by the protocols of *SINTRA*. In other words, current processors are so fast that computationally expensive public-key operations are no longer the dominating element for secure replication protocols.

5 Related Work

Comparing *SINTRA* to other work is difficult because no previous approach provided its strong properties in such a hostile environment. No existing group communication system like ISIS, Horus, or Rampart maintains liveness *and* safety in an asynchronous network with Byzantine faults. The reader is referred to [2] for a detailed comparison with these systems.

The BFT prototype of Castro and Liskov [5] is perhaps the closest in spirit to *SINTRA*. It provides atomic broadcast using an elegant, deterministic protocol, which depends on certain timing assumptions. It is therefore not fully asynchronous, but also requires no public-key operations. Its implementation (in C on Unix) on a LAN performs several orders of magnitude faster than *SINTRA*, taking only a few milliseconds for each atomic broadcast.

The broadcast protocols of Malkhi, Merritt, and Rodeh [11] work in a similar model as *SINTRA*, but implement only consistent broadcast (akin to *SINTRA*'s *consistent channel*). The authors report no performance evaluation for their protocols.

SecureRing [9] and the protocols of Doudou, Garbinato, Guerraoui, and Schiper [7] are examples of protocols that implement atomic broadcast in asynchronous networks augmented with failure detectors for the Byzantine model. But their use of timeouts means they do not qualify as fully asynchronous algorithms, and no performance data has been reported for them.

The only system for which an Internet deployment has been reported in the literature is COCA [19], a secure distributed on-line certification authority. It does not build upon state replication and atomic broadcast, however, and instead uses an application-specific method to impose a partial order on those requests that pertain to the same public key. The average performance of COCA on the Internet (with 1024-bit RSA keys) is reported as 2.3s for a query and 3.7s for an update operation.

6 Conclusion

This is the first demonstration of a secure asynchronous replication protocol on a global scale over the Internet. Although the current *SINTRA* prototype is not extremely fast, it shows that Byzantine fault-tolerant replication is feasible in this environment. Experiments with the prototype show that its performance depends on many details, such as the JVM implementation and the choice of the digital signature scheme. Performance improvements by a factor of two or four seem feasible. Thus, the current implementation of *SINTRA* represents only a worst case and improvements are possible in several aspects:

Choice of programming language: The protocols of *SINTRA* may be implemented in a language for which sophisticated optimizing compilers are available, such as C or C++. Considerable speed improvements seem possible compared to the current Java implementation.

Optimization of the implementation: The current *SINTRA* architecture uses threading heavily, and this seems to be one reason for its slow speed on a LAN. Switching to a thread-per-message model with one or only few threads should yield much better performance. Optimizations are also possible in many other places, for example, in the modular arithmetic of the public-key operations, which is currently done using the standard `BigInteger` operations.

Optimized protocols: The largest performance gain is possible by optimizing at the highest level: the protocols themselves. Recall that *SINTRA*'s atomic broadcast protocol involves Byzantine agreement in every round, even when all servers are honest and answer within a reasonable amount of time. As shown by Castro and Liskov [5] and by Kursawe and Shoup [10], there are so-called *optimistic* protocols that can exploit this and run a much simpler algorithm with one server acting as sequencer in that situation. They switch back to the slower mode of operation when the server is suspected by the others of having failed. This will reduce the cost of atomic broadcast essentially to a single reliable broadcast per delivered message.

Acknowledgments

We are grateful to Victor Shoup for lively discussions about protocols and many contributions to *SINTRA*. Thanks to Hiroshi Maruyama, JR Rao, and Kevin McCurley for hosting *SINTRA*.

This work was supported by the European IST Project MAFTIA (IST-1999-11583), but represents the view of the authors. The MAFTIA project is partially funded by the European Commission and the Swiss Department for Education and Science.

References

[1] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM*, vol. 32, pp. 824–840, Oct. 1985.

[2] C. Cachin, "Distributing trust on the Internet," in *Proc. International Conference on Dependable Systems and Networks (DSN-2001)*, pp. 183–192, 2001.

[3] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols (extended abstract)," in *Advances in Cryptology: CRYPTO 2001* (J. Kilian, ed.), vol. 2139 of *Lecture Notes in Computer Science*, pp. 524–541, Springer, 2001.

[4] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography," in *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 123–132, 2000.

[5] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. Third Symp. Operating Systems Design and Implementation (OSDI)*, 1999.

[6] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[7] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, "Muteness failure detectors: Specification and implementation," in *Proc. 3rd European Dependable Computing Conference (EDCC-3)* (J. Hlavicka, E. Maehle, and A. Pataricza, eds.), vol. 1667 of *Lecture Notes in Computer Science*, pp. 71–87, Springer, 1999.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, pp. 374–382, Apr. 1985.

[9] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing protocols for securing group communication," in *Proc. 31st Hawaii International Conference on System Sciences*, pp. 317–326, IEEE, Jan. 1998.

[10] K. Kursawe and V. Shoup, "Optimistic asynchronous atomic broadcast." Cryptology ePrint Archive, Report 2001/022, Mar. 2001. `http://eprint.iacr.org/`.

[11] D. Malkhi, M. Merritt, and O. Rodeh, "Secure reliable multicast protocols in a WAN," *Distributed Computing*, vol. 13, no. 1, pp. 19–28, 2000.

[12] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1997.

[13] M. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in Rampart," in *Proc. 2nd ACM Conference on Computer and Communications Security*, pp. 68–80, 1994.

[14] M. K. Reiter and K. P. Birman, "How to securely replicate services," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 986–1009, May 1994.

[15] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, pp. 277–288, Nov. 1984.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.

[17] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology: EUROCRYPT 2000* (B. Preneel, ed.), vol. 1087 of *Lecture Notes in Computer Science*, pp. 207–220, Springer, 2000.

[18] V. Shoup and R. Gennaro, "Securing threshold cryptosystems against chosen ciphertext attack," in *Advances in Cryptology: EUROCRYPT '98* (K. Nyberg, ed.), vol. 1403 of *Lecture Notes in Computer Science*, pp. 1–16, Springer, 1998.

[19] L. Zhou, F. B. Schneider, and R. van Renesse, "COCA: A secure distributed on-line certification authority," Technical Report 2000-1828, Department of Computer Science, Cornell University, Dec. 2000.