Append-Only Web Bulletin Board

by
J. Beuchat, jose.beuchat@bluewin.ch
Bern University of Applied Sciences, Switzerland

Supervisor: Dr. E. Dubuis
Date : Feb 21, 2011

Abstract. During the last years, a large number of papers made the assumption that *append-only web bulletin boards* were available. Nevertheless there is no recognized method for the construction of such a bulletin board.
In this paper we identify the requirements of an append-only web bulletin board and present different candidates based on the work of Heather & Lundin and the master's thesis of R.A. Peters. We will see that a Web Bulletin Board alone is not enough. For security reasons and to avoid problems related to hardware, distributed boards and replication of the data are necessary.
Finally, the candidates will be compared and the most suitable solutions will be designated.

Contents

1 Introduction....3
2 The Web Bulletin Board....4
2.1 Requirements....4
2.2 Assumptions....4
2.3 Actors....4
2.4 Applications....5
2.4.1 E-voting....5
2.4.2 Auctions....5
2.4.3 Auditable discussion boards....5
2.4.4 System logs....5
2.4.5 Online Petitions....5
3 Security mechanisms....6
3.1 Threshold Signatures....6
3.1.1 Trivial Threshold Signature....6
3.1.2 Pedersen's Threshold Signature....7
3.1.3 Threshold Signatures Based on Gap Diffie-Hellman Groups....7
3.1.4 Conclusion....8
4 Candidate solutions....9
4.1 Trivial Web Bulletin Board....9
4.1.1 History....9
4.1.2 Reading protocol....10
4.1.3 Writing protocol....11
4.1.4 Conclusion....11
4.2 Synchronized Web Bulletin Board....12
4.2.1 Reading Protocol....12
4.2.2 Writing Protocol....13
4.2.3 Timestamps....14
4.2.4 Deadlocks....14
4.2.5 Conclusion....14
4.3 Unsynchronized Web Bulletin Board....14
4.3.1 Reading Protocol....15
4.3.2 Writing Protocol....15
4.3.3 Conclusion....15
4.4 Krummenacher's Web Bulletin Board....16
4.4.1 Reading protocol....16
4.4.2 Writing protocol....16
4.4.3 Conclusion....17
4.5 Peter's Web Bulletin Board....18
4.5.1 Protocol Layers....18
4.5.2 Secure Group Membership Protocol....19
4.5.3 Rampart Protocol....22
4.5.4 Synchronized Atomic Multicast Protocol....24
4.5.5 Reading Protocol....25
4.5.6 Writing Protocol....25
4.5.7 Conclusion....26
5 Evaluation....28
6 Conclusion....29

1 Introduction

More and more data are published on the Internet everyday. Those are accessible to everyone but how can we ensure that the displayed content hasn't been modified? In serious contexts (e.g. e-voting) it is essential to prove that the published data haven't been altered.

Using a *Web Bulletin Board*, we can ensure that the users will be able to post messages and have the assurance that they will never be changed, moved or deleted. Also, the messages will be available to everybody and every authorized user will be able to post.

Unfortunately, even if the existence of the *Web Bulletin Board* is broadly assumed, the information about its requirements or a working solution remain poor.

The goals of this paper are to identify the requirements of a *Web Bulletin Board*, to determine candidate solutions and to compare them. This will be realized on the basis of the two following papers: *The Append-Only Web Bulletin Board* [1] by Heather & Lundin and *A Secure Bulletin Board* [2] by Peters. In the first one, a trivial solution is clearly defined and hints about two distributes boards are given. In the second paper, a board using existing protocols is described.

This report is organized as follows: first, the basic requirements, actors and possible applications of a *Web Bulletin Board* are identified. Section 3 contains security mechanisms used by the candidate solutions presented in Section 4. Then, a comparison between the different candidates is done and finally, Section 6 is the conclusion.

2 The Web Bulletin Board

A *Web Bulletin Board* is responsible for publishing something (not necessary in e-voting context) and giving the proof that its content has been altered.

2.1 Requirements

1. **Availability**. Each allowed user is able to successfully send messages to the board. Similarly, everybody should be able to read and test the board.
2. **Unalterable History**. Once a message is published it should never be **removed** or **modified**. Moreover, no message should be **moved** to another position and the new ones should be placed **at the end**. If those properties are not respected, the users are able to **prove it**.
3. **Certified Publishing**. If a *Reader* retrieves the content of the *Web Bulletin Board*, he will have the **proof** for each message of **who** posted it and that the *Writer* intended the message to be published with the stated **timestamp** and at **this point** in the board's sequence of messages.

Those requirements should stay even if the *Web Bulletin Board* and a writer **collude**.

2.2 Assumptions

Many assumptions are needed in order to achieve our security requirements.

Assumption 21. ***Digital signatures*** *can be verified by everybody but only the* ***owner*** *of the private key is able to produce them.*

Assumption 22. *A solution to* ***publish*** *the* ***public keys*** *exist (e.g. PKI) so that all agents know the public keys of all writers and all Web Bulletin Boards, but* ***secret keys*** *remain* ***private.***

Assumption 23. *Each party is able to* ***generate*** *its own* ***key pair****.*

Assumption 24. *In the case where a threshold signature scheme is used, we assume that a* ***threshold key generation protocol*** *is included.*

Assumption 25. *Communication between different parties is done using security mechanisms which, when necessary, provide* ***privacy****,* ***authenticity*** *and* ***integrity****.*

Assumption 26. *If* ***hash functions*** *are used, they are* ***collision-free****. Two distinct terms will never result to the same hash value.*

Assumption 27. *The Web Bulletin Board is responsible for ensuring that the* ***metadata*** *(signatures, hashes, ...) are* ***correctly*** *generated by the writers.*

2.3 Actors

Three types of actors exist. Some of them are active (modifying the data) and some are not:

Web Bulletin Board. The Web Bulletin Board is the history. It is responsible for allowing the **writers** to publish **information** on the board and making them accessible to any of the **readers** but is not allowed to post. The content of the **messages** (data and metadata) is **verified** by the board itself before being added to its history. The *Web Bulletin Board* is also responsible for ensuring that its **content** does not change and being able to **detect** it if it does.

Writer. The writer is active. He is the only user allowed to **post** information on the *Web Bulletin*

Board and is responsible for **generating** the necessary **metadata** (e.g. digital signature) to be sent to the board. In an e-voting context he is the **voter**.

Reader. The reader isn't active but is still very important. Each time he reads the data, the validity of the history is tested. Thus, the more readers there are, the more the content of the board is tested and the better the security is.

2.4 Applications

2.4.1 E-voting

Voters want to be sure that their votes have been counted correctly and that they have the possibility to revoke the vote otherwise. A *Web Bulletin Board* is responsible for publishing the encrypted ballots and providing a receipt. At the end the votes are decrypted and published (without anyone knowing the link from encrypted to decrypted votes). It is then possible to count the decrypted ballots. To ensure that both parts correspond without linking them, *zero-knowledge proofs* described in *[3]* are used. If a ballot is missing, the voter can prove it by giving his receipt.

2.4.2 Auctions

A bidder who wishes to place a bid wants to receive a proof for it, otherwise anyone could refute his bid. The sequence is very important here.

2.4.3 Auditable discussion boards

For many reasons, it could be interesting to provide a forum with a secure history.

2.4.4 System logs

Logs are system activities written in text files. It is useful to have security here if we do not trust the logger.

2.4.5 Online Petitions

In the context of online petitions, security could be needed, for example, to prove that the text of a petition hasn't been changed.

3 Security mechanisms

3.1 Threshold Signatures

Threshold signatures play an important role in the candidate solutions presented in Section 4.

The goal of a signature is to authenticate data. In distributed solutions (using more than one party), the receipt (authentication of the data) is generated using a *Threshold Signature* scheme. The difference with a normal signature is that several parties have to cooperate to generate a valid signature. In a (t,n)-threshold signature scheme, the private key is shared between n parties so that t of them can jointly produce a valid signature. A *Threshold Set* is made of t parties working together.

The three solutions presented here are the *Trivial Threshold Signature*, the *Pedersen's Threshold Signature* described in [4] and the *Threshold Signatures Based on Gap Diffie-Hellman Groups* described in [5].

The main points are the key generation, the signature share construction, the signature share combination and the signature validation.

3.1.1 Trivial Threshold Signature

In the *Trivial Threshold Signature*, the signature is a composition of t basic signatures, like those used in non-distributed systems.

Key Generation

The keys are generated by the n parties themselves. They are responsible to create their own key pairs and to publish their public keys. We do not use a dealer to generate and distribute the keys, since it would create a *Single Point Of Failure*[1].

Signature Share Construction

Each party signs the data to authenticate with its private key. A share here, is a signature like those used in non-distributed systems.

Signature Share Combination

The combination here is nothing more than a list of shares and their corresponding public keys. The signatures shares are regrouped, tested and inserted in the list. The receipt consists of x different signatures, where $t \leq x \leq n$. The disadvantage is that the size of the signature is linear in the number of participants.

Signature Validation

To validate the threshold signature, the shares have to be verified one by one using the public keys. Once we tested at least t shares, we have the assurance that at least t parties signed the message. Of course, a share should be only once in the list.

1 *Part of a system which, if it fails, will stop the entire system from working.*

Conclusion

This solutions meets our requirements and is easy to implement but the size of the threshold signature raises linearly with n.

3.1.2 Pedersen's Threshold Signature

Pedersen's Threshold Signature scheme[6] is based on Schnorr signatures[7] and Feldman's Verifiable Secret Sharing Scheme (VSSS) [8].

Key Generation

Each party runs Feldman's VSSS [8] protocol to share a secret. Then, the party uses his share to compute its own part of the private key and all the shares together compose the public key of the *Web Bulletin Board*.

Signature Share Construction

Each party runs Feldman's VSSS protocol to generate a random global value known by everyone involved in the protocol. Then, each party computes its own part of the signature (signature share).

Signature Share Combination

Using the t signature shares, it is possible to compute the threshold signature. That means that there will be only one signature in the receipt.

Signature Validation

Test the signature using the public key. If it is valid, we have the assurance that at least t parties signed the message.

Conclusion

This solution meets our requirements. There is no trusted dealer and the receipt contains one single signature, which is very small. This is a big advantage in comparison to the *Trivial Threshold Signature* scheme presented in 3.1.1. This scheme is also considered as easy to implement.

3.1.3 Threshold Signatures Based on Gap Diffie-Hellman Groups

This scheme, presented by Boldyreva, uses elliptic curves. For details, please refer to [5].

Key Generation

Each parties runs Feldman's VSSS to share a secret[7]. Then everyone involved in the protocol will have a x_i and a global $y = g^x$ where x is determined by each t out of n parties.

Signature Share Construction

Given a message m, each party constructs the share $s_i = H(m)^{x_i}$.

Signature Share Combination

Using t valid shares $s_i \in A$, compute the threshold signature $s = \Pi_{i \in A} s_i \lambda_{Ai}$.

Signature Validation

Test the signature using the public key. If the signature is valid, we have the assurance that at least t parties signed the message.

Conclusion

This solution meets our requirements. There is no trusted dealer and the receipt contains one signature. The implementation is considered as complicated.

3.1.4 Conclusion

The three schemes presented here meet our requirements.

The *Trivial Threshold Signature* scheme is very easy to implement and to understand. Also it is the only one that does not need a key generation protocol using a communication channel or collaboration with other parties. Thus, it is perfect to be used in the set-up phase of the *Web Bulletin Board*. However the receipt can be very large and performances are not optimal.

The *Pedersen's Threshold Signature* contains only signature but the performances again are not optimal.

Threshold Signatures Based on Gap Diffie-Hellman Groups contain one signature and provide good performances. However, this scheme is complicated to implement.

4 Candidate solutions

In this Section we make the assumption that every party is honest and acting correctly. Five solutions are presented with their main properties. Their strengths and weaknesses are analyzed.

4.1 Trivial Web Bulletin Board

This solution is clearly defined by Heather & Lundin in [1]. It consists of one board which directly communicates with all the *Writers* and *Readers* as represented in *Figure 1*.

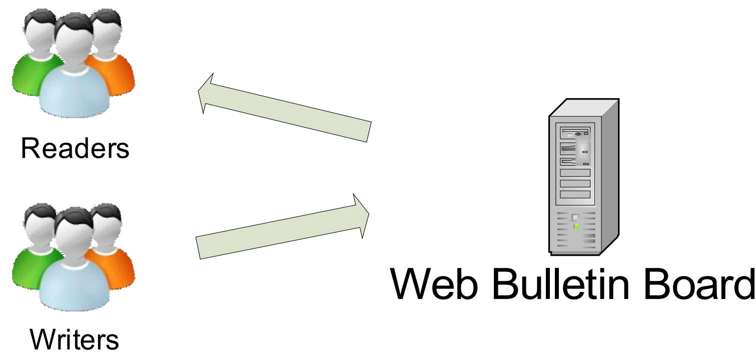

Figure 1: Trivial Web Bulletin Board architecture.

The *Web Bulletin Board* is public to everybody but only authorized *Writers* are allowed to post messages.

4.1.1 History

The board stores a sequence of entries <wbb_1, …, wbb_n>. Those entries, also called the *history*, contain the messages and the necessary metadata. Table 4 1 is a list of useful abbreviations and notations. Table 4 2 represents the *history*:

B:	Board.
W:	Writer.
R:	Reader.
m:	Message.
T:	Timestamp at the time the *Writer* writes the message.
T':	Timestamp at the time the board signs the message.
H(.):	Hashing function.
H:	The hashed concatenation of *m*, *T*, *W* and H_{i-1}. $H(m_i.T_i.W_i.H_{i-1})$. Note that H_0=0.
H_λ:	The last entry of the history, also called the *state* of the board.
S(.):	Signing function.
WSign:	The hash value signed by the *Writer:* $S_{wi}(H_i)$.
BSign:	The signed concatenation of *WSign* and *T'*: $S_B(WSign_i.Ti')$.

Table 4 1: Useful abbreviations and notations.

m_i	T_i	W_i	H_i	$WSign_i$	$BSign_i$
m_1	T_1	W_1	$H(m_1.T_1.W_1.0)$	$S_{W1}(H_1)$	$S_B(WSign_1.T_1')$
m_2	T_2	W_2	$H(m_2.T_2.W_2.H_1)$	$S_{W2}(H_2)$	$S_B(WSign_2.T_2')$
...	...	...	...	...	...
m_n	T_n	W_n	$H(m_n.T_n.W_n.H_{n-1})$	$S_{Wn}(H_n)$	$S_B(WSign_n.T_n')$

Table 4 2: History of the Board presented by Heather & Lundin.

We can see that H_i is made of data coming from the actual entry as well as the *H* from the previous entry. That leads to a **concatenation** of the data. Thus, new messages can only be added **at the end** of the list and if some of them are modified, moved or removed, it will easily be detected by anyone reading the *history*. That was the second requirement (Section 2.1).

A security parameter ε is introduced to define a maximal amount of time (some ms) during which the messages can be received by the *Writer* and published. If the board receives the message more than ε time after the *Writer* wrote it, it will be rejected. Furthermore, if a *Reader* reads the history at time *T* and later a message with a timestamp T_i smaller than $T - \varepsilon$ appears, that means that the board has been corrupted. That was part of the third requirement.

The *history* can be tested by anyone at anytime. It is called **consistent** if the following points are true:

1. $H_i = H(m_i.T_i.W_i.H_{i-1})$
2. $WSign_i = S_{Wi}(H_i)$
3. $BSign_i = S_B(WSign_i.Ti')$
4. $T_i \leq T_i' < T_i + \varepsilon$

4.1.2 Reading protocol

The information on the board are public and thus everybody is allowed to read them. The communication over the Internet (e.g. HTTP protocol) or the fact that for efficiency reasons, readers may want to retrieve only part of the messages does not concern us here. We assume that the whole list of messages is transferred.

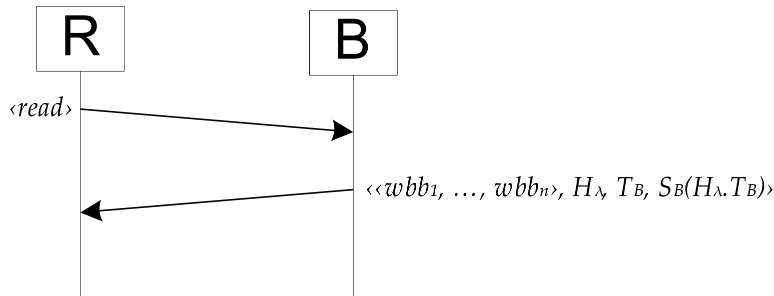

Figure 2: Reading protocol of the Trivial Web Bulletin Board.

1. R asks the content of the board in the **read** message.
2. B answers with its history and a proof that it is consistent and up to date.

The *Reader* can now test the consistence of the *history.* If later the *history* changes (e.g. a message is removed at the end), he can prove that the board is corrupted by giving the signature $S_B(H_\lambda.T_B)$.

4.1.3 Writing protocol

The protocol executed between the *Writer* and the *Web Bulletin Board* is presented in Figure 3. We make the assumption that the digital signatures are always tested and if they are wrong, the process is stopped.

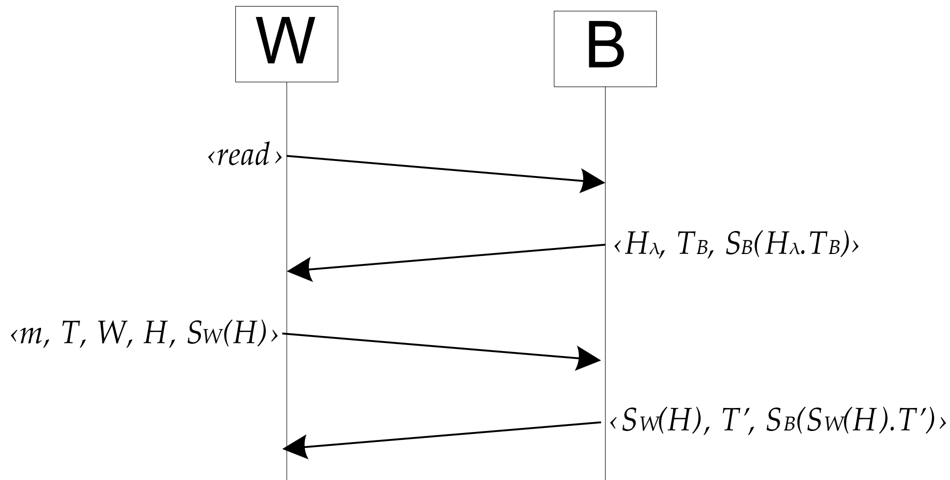

Figure 3: Writing protocol of the Trivial Web Bulletin Board.

1. The *Writer* asks the status of the board.
2. B sends a signature of its status H_λ and a timestamp to W.
3. W tests if $T-T_B>\varepsilon$. If not, he sends his message and the necessary metadata to B.
4. B tests if $T'-T>\varepsilon$. If not, it adds m to the history and sends the receipt to W.

The *Writer* finishes by testing if the receipt is well formed and makes sure that is message m has been added to the *history*. If not, he knows that the board is corrupted. He also has to store both messages received from the board, so that he later can prove that his message was published at this position in the *history*.

If more than on *Writer* send a message at the same time, only the first one will be published. The second will be considered as badly formed and rejected, since the H_λ does not correspond anymore.

4.1.4 Conclusion

The *Trivial Web Bulletin Board* is made of one single board. Indeed if this one is corrupted or unavailable (hardware problem, DOS[2] attack, …), the whole system is unusable. There is a *Single Point Of Failure* problem. We therefore can say that the first requirement (availability) is not satisfied.

2 *Denial Of Service: attempt to make a computer resource unavailable to its intended users.*

This solution is based on mathematical algorithms. In other words, we do not need to trust the *Web Bulletin Board* if enough *Writers* make sure that their messages are correctly published. The *Readers* can also give more confidence by reading the content of the *history* and proving it if something has changed. That is our second requirement (unalterable history).

The *Reader* who retrieves the list of messages will get something similar to Table 4 2. He thus will know for each message, who wrote it, when and at which position. That satisfies the third requirement (certified publishing).

The performances are good, since only one board is used and the amount of data transferred and tested is very small.

The *Trivial Web Bulletin Board* is easy to understand and implement.

4.2 Synchronized Web Bulletin Board

To eliminate the *Single Point Of Failure* problem seen in the *Trivial Web Bulletin Board*, Heather & Lundin propose to distribute the board in n parties of whom at most k can fail. However, their description is very poor. In this Section, I present my own interpretation of a *Synchronized Web Bulletin Board*.

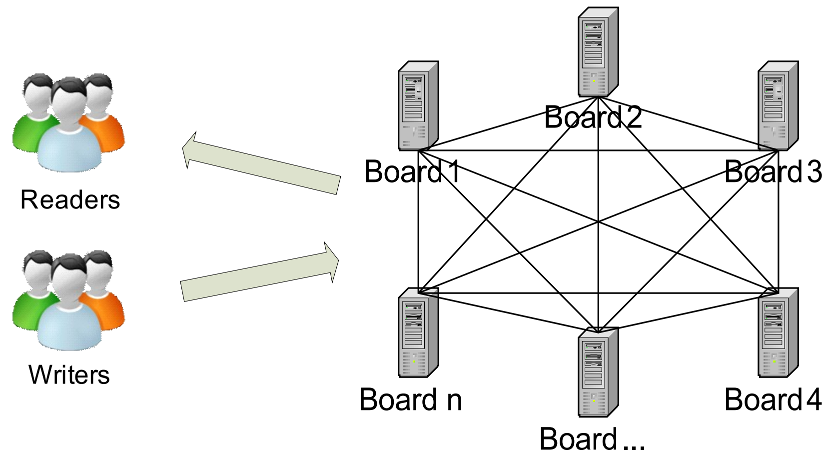

Figure 4: Synchronized Web Bulletin Board architecture.

Since there are more than one board and that some of them are possibly corrupted, we use *Threshold Signatures*, presented in Section 3.1. The size of a threshold set is $t \geq k+1$, so that if k boards are corrupted, they can not generate a signature. Also, to avoid two messages to be published at the same time, t has to be bigger that the half of n. $2t > n$.

4.2.1 Reading Protocol

During the reading protocol, one party is contacted. It asks t-1 others to generate a signature share, confirming the state of the board, and sends it back to the *Reader*. Figure 5 describes the protocol more in details.

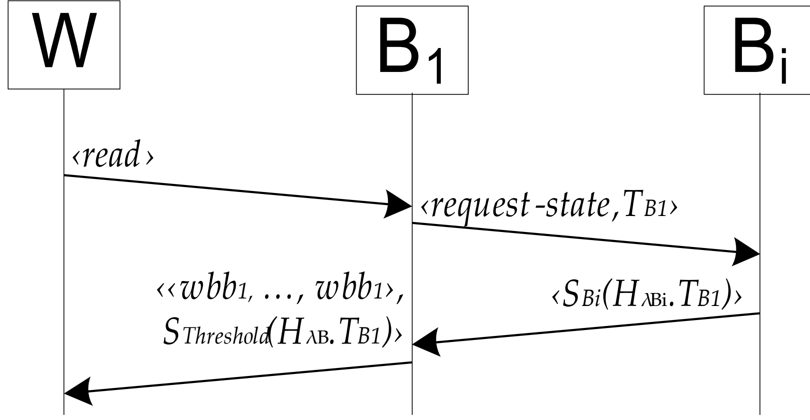

Figure 5: Reading protocol of the Synchronized Web Bulletin Board. B_i:B_2...B_t.

1. The *Reader* asks the content of the boards to B_1.
2. B_1 asks *t* other boards to generate a signature share, using T_{B1} and their status. T_{B1} has to be sent so that the boards agree on an identical time value.
3. The boards B_i, receiving this request, test if $T_{Bi} - T_{B1} < \varepsilon$. If so, they return a signature share containing T_{B1} and their status $H_{\lambda Bi}$.
4. Since the boards are synchronized, the status $H_{\lambda Bn}$ should be the same everywhere. The shares can thus be assembled by B_1. The receipt containing the messages and a threshold signature is returned to *R*.The reader can now test if the *history* is well formed by reconstructing H_λ.

4.2.2 Writing Protocol

During the writing protocol, the *Writer* randomly chooses one of the *n* boards (say B_1), which asks *t*-1 other parties to generate a signature share and publish the message. At the end, the message is sent to the *n-t* other parties for publication and a receipt is sent to the *Writer.* The receipt attest that *t* of *n* boards published the message. This protocol represented in Figure 6.

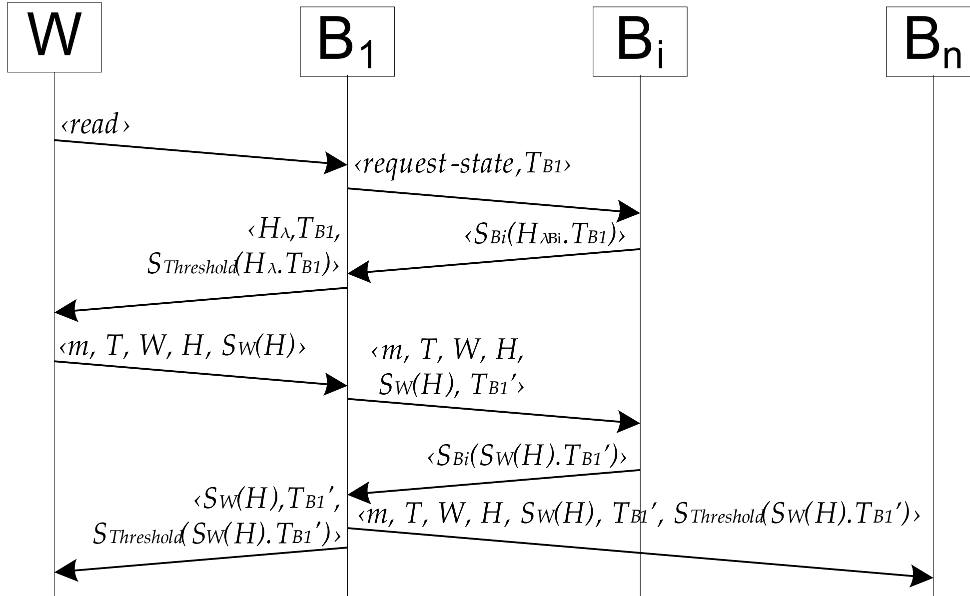

Figure 6: Writing protocol of the Synchronized Web Bulletin Board. B_i *is the threshold set:* $B_{2...t}$. B_n: $B_{t+1...n}$.

1. The *Writer* asks the status of the boards. This is done similarly to the *Reading Protocol*.
2. After receiving the status, *W* sends his message *m* with the necessary metadata.
3. B_1 request a signature share for this message, providing T_{B1}' to B_i. Again, this has to be done in order to get signature shares of the same content.
4. The *t-1* other boards generate the shares and send them to B_1, which combine them into a Group-threshold signature (Section 3.1).
5. B_1 sends the message to the other boards and returns the receipt to *W*.

The *Writer* tests if the receipt is well formed and makes sure that is message *m* has been added to the *history*. He also has to store both messages received from the board, so that he later can prove that his message was published at this position in the *history*.

4.2.3 Timestamps

This approach uses timestamps, which have to be trusted by each party. Therefore we need an external time server. Its only role is to send signed timestamps. Again, there is a risk of *Single Point of Failure*. This problem would have to be carefully examined. Solutions exist but are not developed here.

4.2.4 Deadlocks

In the protocol described in Figure 6, there is a risk of deadlocks and livelocks. If two *Writers* start the protocol at the same time, two boards will ask different parties to sign and publish a message at the same position and will never get the *t* signatures share they are expecting. Of course solutions exist to avoid deadlocks (e.g. timers), but they are difficult to implement and costly. Furthermore, *Writers* who encounter this problem will have to restart the process and a corrupted party could find a way to make the system unusable.

4.2.5 Conclusion

Using *n* boards, the *Single Point Of Failure* present in the *Trivial Web Bulletin Board* disappears and the system does not stop even if some parties stop working correctly. The first requirement (availability) is thus satisfied.

Since we use the same *history* as in the *Trivial Web Bulletin Board*, we know that unalterable history and certified publishing are also provided.

The *histories* are the same at each board. Thus, the messages have the same order everywhere.

Group-threshold signatures are used, reducing the amount of data to be transferred and tested.

However deadlocks are problematic and even if solutions exist, their complexity and cost represent a problem.

4.3 Unsynchronized Web Bulletin Board

Similarly to the *Synchronized Web Bulletin Board*, the *Unsynchronized Web Bulletin Board* is distributed in *n* parties of whom at most *k* can fail.

However, here, the size of the threshold set (*t*) does not have to be bigger than $n/2$ anymore. That gives now the possibility to create several threshold sets and to write several messages at the same time. The consequence is that the *histories* are no longer the same everywhere. To retrieve all the messages, we now have to read the content of at least n-t+1 boards.

The fact that the *histories* can be different leads to another problem: which value of H_λ should the *Writer* use to construct his message? Heather & Lundin do not give any hint about that. The solution presented below is based on my own interpretation.

4.3.1 Reading Protocol

In their paper, Heather & Lundin say that the *Reader* has to read at least n-k+1 *histories*. Leaving k-1 boards unchecked is not a problem, because we know that a message is on at least one of the n-k+1 other boards.

However, here, because we can not use Group-threshold signatures (reason explained in 4.3.2), the messages have to be present on more than k boards to be counted. Indeed, if corrupted boards publish fake messages, we have no other way to prove it. Thus, to get a consistent *history*, we will have to read the n boards.

The basis of the reading protocol is similar tho the one in the *Trivial Web Bulletin Board* (Section 4.1.2).

4.3.2 Writing Protocol

Since the H_λ are different in each *history*, the *Writer* has to create a different message for each board. In this solution, the protocol is started independently on i boards. The writing protocol is exactly the same than the one presented in the *Trivial Web Bulletin Board* (Section 4.1.3).

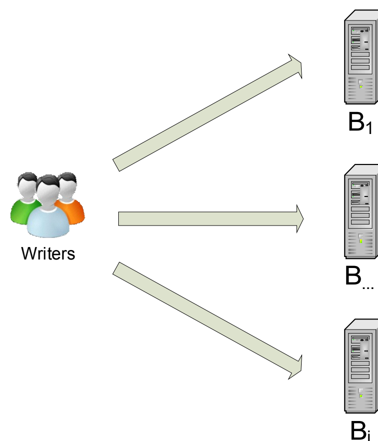

Figure 7: Writing protocol of the Unsynchronized Web Bulletin Board.

The value i should be bigger or equal to the threshold set t, so that W obtains t different signatures shares to create the threshold signature. Thus, $k < t \leq i \leq n$.

If a corrupted board does not runs the protocol correctly, the *Writer* is responsible to start it on another board. This problem is known as the *Byzantine agreement*[9]. The boards independently publish the messages without knowing if the other will do the same.

As the status of the boards (H_λ) are different everywhere, Group-threshold signatures are not possible. Therefore we must use trivial threshold signatures (Section 3.1.1) with a different private key for each board.

4.3.3 Conclusion

The *Web Bulletin Board* is distributed in n parties and no *Single Point of Failure* exists. Thus, the first requirement (availability) is fulfilled.

Even if the content of the *history* is not identical at every party, the structure is the same than in the *Trivial Web Bulletin Board* (see Table 4 2). Thus, the second and third requirements (unalterable history and certified publishing) are fulfilled.

We can prove that no message has been moved, however, the *histories* can be different at each party and giving a final order to the messages is impossible. This solution is thus not usable in every cases (Section 2.4.2).

The parties do not communicate with each other and do not sign the same data. Thus, Group-threshold signatures are not possible.

The responsibility to get enough signatures is on the *Writer* side. The board can not know if enough parties are ready to publish the message and thus, if it should publish it or not. There is no *Byzantine agreement*.

Since the protocols are similar to those presented in the *Trivial Web Bulletin Board*, this solution can be considered as easy to implement.

4.4 Krummenacher's Web Bulletin Board

This solution, proposed in [10], is also based on the work of Heather & Lundin. It consists of a *Web Bulletin Board* distributed in n parties of whom at most k can fail. Each party has its own key pair and a part of a key used to generate threshold signatures. Threshold value is $t \geq k+1$.

In his paper, Krummenacher introduces the possibility that a *Web Bulletin Board* contains more than one *history*. Indeed if *Web Bulletin Boards* of different groups of interest are used, that reduces their profit to conspire. Also, if each of them contains several *histories*, that reduces the number of necessary *Web Bulletin Boards.*

4.4.1 Reading protocol

Similarly to the *Unsynchronized Web Bulletin Board* (Section 4.3), the content of the boards are not the same everywhere. However, because we use Group-threshold signatures, we do not have to test if the messages have been published on at least k+1 parties. Thus, the *Reader* has to read at least n-k+1 *histories*.

The reading protocol is the same as the one presented in the *Trivial Web Bulletin Board* (Section 4.1.2).

4.4.2 Writing protocol

In this solution, the *Writer* starts the protocol on l boards, where $l \geq t$. In Figure 8, describing the writing protocol, the *histories* are differentiated using an ID: ID_H. The messages also have an ID: ID_M and the boards identities are in a list <*1,...,l*>.

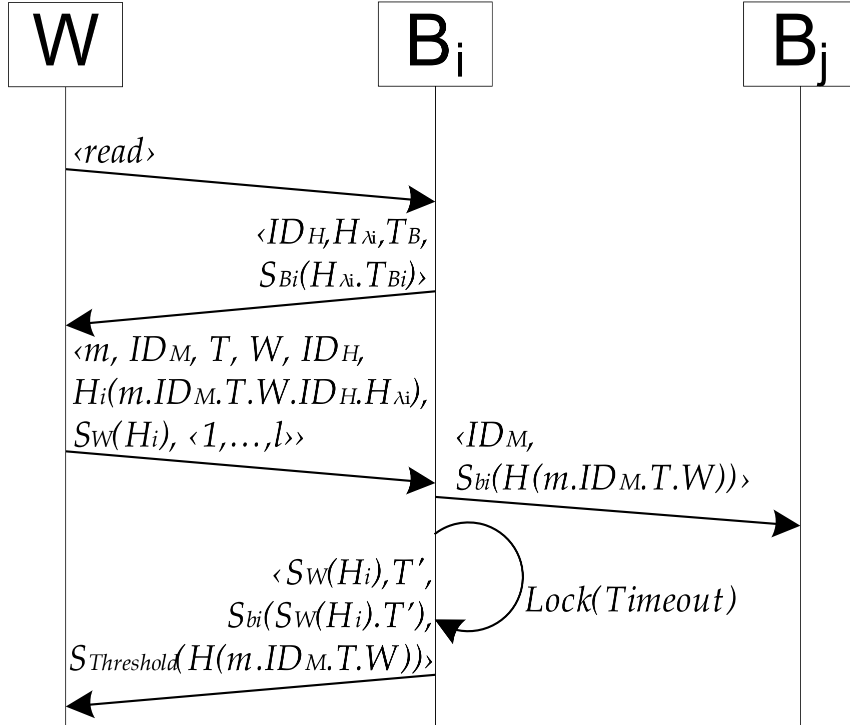

Figure 8: Writing protocol of Krummenacher's Web Bulletin Board. B_i:$B_{1...l}$ *and* B_j:$B_{1...l}$, $j \neq i$.

1. *W* asks the status of *l* different boards.
2. B_i answers with its status and a timestamp.
3. *W* sends his message and the corresponding metadata to the B_i.
4. After the reception of the previous message, B_i sends to the *l*-1 other boards an attestation, telling them that it will publish the message *m* if *t*-1 other boards to the same. Then it locks itself. Using a timeout after the fourth message eliminates possible deadlocks.
5. If the board B_i does not receive *t*-1 attestations after some milliseconds, it releases the lock and nothing is published. If it receives them, the message is published, the lock released and the receipt containing the threshold signature is returned to *W*.

The fact that the boards wait to receive t-1 attestations before publishing the messages solves the *Byzantine agreement* problem.
ID are used here, so that, the boards know with who they have to communicate.

4.4.3 Conclusion

Krummenacher's Web Bulletin Board is distributed and no *Single Point of Failure* is possible. Its availability is thus assured and as in the *Trivial Web Bulletin Board*, unalterable history and certified publishing are also provided.

Furthermore the system attests that a message is published on at least *k*+1 boards or not at all and that the *Writer* gets a receipt only if his message has been published on at least *k*+1 boards. There is a solution to the *Byzantine agreement*.

Group-threshold signatures are used, reducing the amount of data to be transferred and tested.

A solution to eliminate deadlocks has been found (using timers).

The order of the messages is not necessary the same on every board even if timestamps are used.

The implementation of this solution can be considered as affordable.

4.5 Peter's Web Bulletin Board

This solution is based on the secure bulletin board of R.A. Peters [2]. It is an asynchronous distributed protocol executed by several parties that tolerates the corruption of at most $\lfloor (n-1)/3 \rfloor$ group members.

Since the bulletin board is composed of many parties, in addition to secure communications, we need to be sure that if a message is received by one party, every other party receives the same message in the same order. Furthermore, this has to stay true even if some parties are corrupted.

This problem is known as the *Byzantine agreement*[9]. The solution proposed here is a distributed protocol based on [11].

Peter's Web Bulletin Board uses together the *Secure Group Membership Protocol* (Section *4.5.2),* the *Rampart Protocol* (Section 4.5.3) and *Threshold Signatures* (Section 3.1).

The *Secure Group Membership Protocol*, gives the possibility to add or remove a party. Thus if a board is corrupted or suffers from hardware problems, it can be removed and a new one can be added.

4.5.1 Protocol Layers

As represented in Figure 9, each party runs a protocol organized in layers.

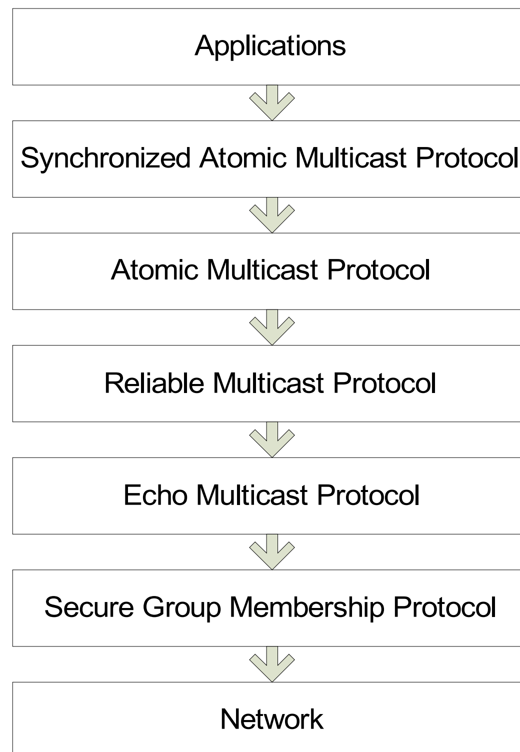

Figure 9: Protocol layers.

Network. The network is responsible for the communication between the different parties. It is

considered authenticated and confidential.
Secure Group Membership Protocol. It establishes a group view (set of supposed correct members) and assures that it is the same at each honest party. It is also responsible for adding and removing processes and delivering the corresponding new group views.
Echo Multicast Protocol. It enables a party to multicast a message to a certain view and ensures that every honest party of this view delivers the same message.
Reliable Multicast Protocol. It ensures that all honest group members receive the same messages, even in the face of malicious multicast initiators and ensures common group views.
Atomic Multicast Protocol. It adds the property that honest members deliver these messages in the same order.
Synchronized Atomic Multicast Protocol. It ensures that members that have been offline for a moment are able to synchronize all the messages.
Applications. This layer represents the applications using the bulletin board. For example, a voting application that receives votes from clients and store them in a list.

4.5.2 Secure Group Membership Protocol

In this Section we describe the *Secure Group Membership Protocol* [12], a solution for **asynchronous distributed systems** that tolerates the malicious corruption of group members. This protocol gives the assurance that in a distributed system, the honest parties agree on a set of currently operational members. It is possible to **invite** or **remove** parties in the group. Furthermore, a malicious party cannot effect changes to the group or prevent a change to be done.

Each party p_i has a **view** V_i^x which consists of the **correct** parties. The view can change and each time it does x is incremented, representing the x-th view. Initially, each party is in view V^0, which is manually configured at each member. The protocol also ensures that each view is the same at each honest party and that p receives V^x before V^y if $x < y$ and p is in both.

For the correctness of this protocol we assume that at least $\lceil (2|V^0|+1)/3 \rceil$ parties are correct.

This protocol is not concerned with the detection of corrupt group members but to remove them from the group once detected.

Properties

The protocol satisfies the following four properties.

Uniqueness. If p_i and p_j are correct and V_i^x and V_j^x are defined, then $V_i^x = V_j^x$. That means that the x-th view is the same for every group member.

Validity. If p_i is correct and V_i^x is defined then $p_i \in V_i^x$ and for all correct $p_j \in V_i^x$, it holds that V_j^x is eventually defined.

Integrity. If $p_i \in V^x \setminus V^{x+1}$, then *faulty($p_i$)* held at some correct $p_j \in V^x$, and if $p_i \in V^{x+1} \setminus V^x$, then *correct($p_i$)* held at some correct $p_j \in V^x$. This property prevents a view change to occur due to one single party.

Liveness. If there is a correct $p_i \in V^x$ such that $\lceil (2|V^x|+1)/3 \rceil$ correct members of V^x do not suspect p_i *faulty*, and a $p_j \in V^x$ such that *faulty(p_j)* holds at $\lfloor (|V^x|-1)/3 \rfloor + 1$ correct members of V^x, then eventually V^{x+1} is defined. Similarly, if there is a member $p_j \notin V^x$ and *correct(p_j)* holds at $\lfloor (|V^x|-1)/3 \rfloor + 1$ correct members of V^x, then eventually V^{x+1} is defined. That means that if enough correct members agree about removing or adding a process, the membership is eventually changed.

Protocol

A total order is assumed to exist on the parties. In each V^x there is a distinguished member called the **manager**, which has the highest rank. The manager is responsible for **suggesting** an update to the view. His suggestion is based on the recommendation of group members. The protocol we consider here is represented in Figure 10.

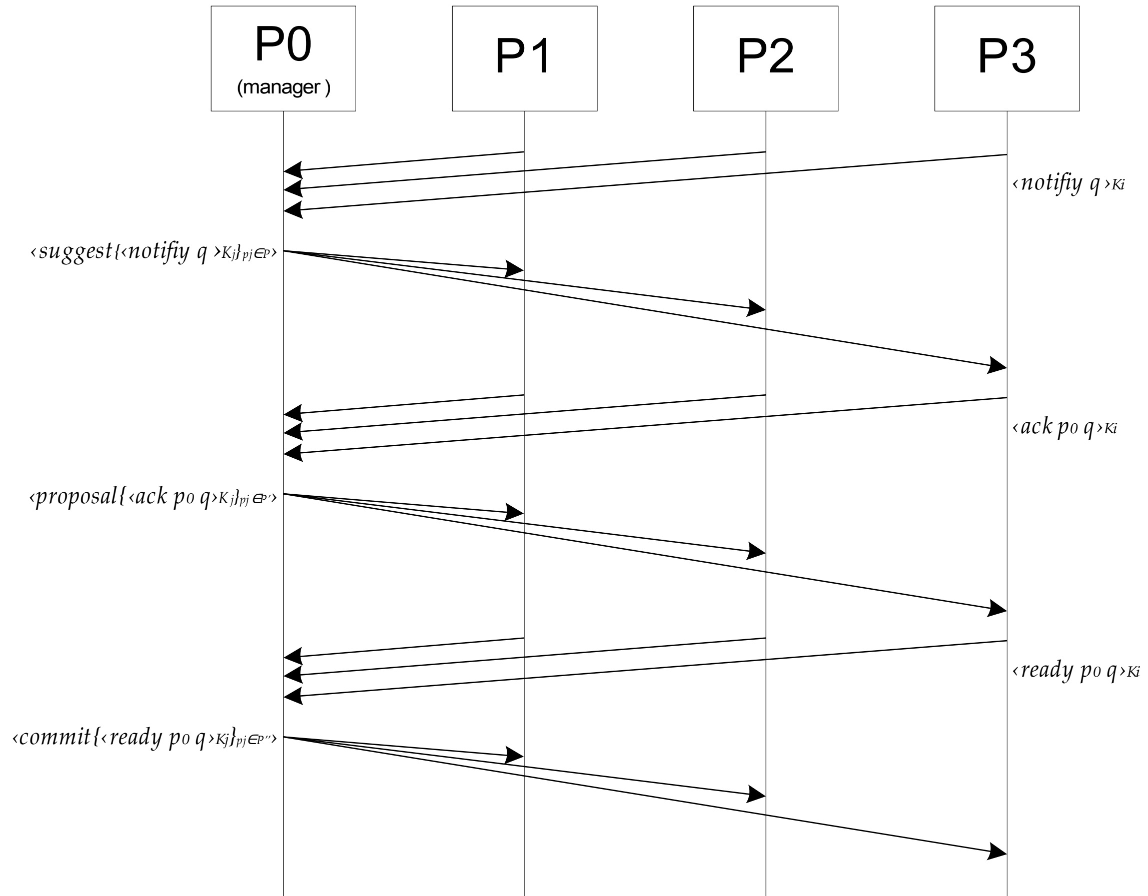

Figure 10: Protocol when the manager is correct.

The protocol executes as follow:

1. The group members suspecting a party q to be faulty send $\langle notify\, q \rangle_{K_j}$ to the manager.
2. The manager, say p_i (or p_0 in Figure 10), collects the notification until it has $\lfloor (|V_i^x|-1)/3 \rfloor + 1$ of them. If that happens, it sends a suggestion $\langle suggest\{\langle notify\, q \rangle_{K_j}\}_{p_j \in P} \rangle$ to the members of V_i^x .
3. When each party p_j receives this message from the manager, it returns a signed acknowledgment $\langle ack\, p_i q \rangle_{K_j}$ to the manager.
4. Once the manager has $\lceil (2|V_i^x|+1)/3 \rceil$ acknowledgment, it sends a proposal $\langle proposal\{\langle ack\, p_i q \rangle_{K_j}\}_{p_j \in P} \rangle$
5. When p_j receives the proposal, it verifies it and returns $\langle ready\, p_i q \rangle_{K_j}$ indicating its readiness to commit the update.
6. Once p_i collects $\lceil (2|V_i^x|+1)/3 \rceil$ **ready** messages it broadcasts a **commit** message $\langle commit\{\langle ready\, p_i q \rangle_{K_j}\}_{p_j \in P} \rangle$

7. A party p_j that receives this message, installs the new view V_j^{x+1} by adding or removing q.

If the manager is suspected to be faulty, some party, called a **deputy**, may need to take over the manager. A party p_i which is not the manager, becomes a **deputy** if enough members suspect all other members with a higher rank of being faulty.

1. If a process p_j suspects all members with rank higher than p_i it sends a message $\langle deputy\, p_i \rangle_{K_j}$ to p_i.
2. If p_i receives $\lfloor (|V_i^x|-1)/3 \rfloor + 1$ messages it initiates the deputy protocol by broadcasting $\langle query \{ \langle deputy\, p_i \rangle_{K_j} \}_{p_j \in P} \rangle$ to the members of V_i^x. This messages proves that some correct members believes that p_i should become a deputy.
3. In response, each member p_j returns $\langle last\, p_i\, S \rangle_{K_j}$ where S is the set of acknowledgments contained in the last proposal sent by the previous manager or Ø if it has not yet received a proposal. The set S is returned to convey any update that could have been committed by the manager or a deputy of a higher rank.
4. Upon receiving $\lceil (2|V_i^x|+1)/3 \rceil$ last messages $\{ \langle last\, p_i\, S_j \rangle_{K_j} \}_{p_j \in P}$, p_i sends a suggestion $\langle suggest \{ \langle last\, p_i\, S_j \rangle_{K_j} \}_{p_j \in P} \rangle$ to V_i^x.
5. Each party forms the new view by removing the party suggested by the party with the lowest rank.

A corrupted manager may try to convince one party of one update to the group view, while having another party forming another view. Before a manager can propose a party to be removed, it needs $\lfloor (|V_i^x|-1)/3 \rfloor + 1$ signed requests, so at least one honest party wants p to be removed. The manager then forms a proposal, using $\lceil (2|V_i^x|+1)/3 \rceil$ signed **ack** responses.
Once a party signed a **ack** message, **it refuses to sign ack** responses for other parties. Since each honest party only signs one proposal, and since a proposal needs $\lceil (2|V_i^x|+1)/3 \rceil$ **ack**, a manager can form at most one proposal.
We can think that a party receiving a correct **proposal** message has enough information to update the current view. Only one correct **proposal** can be formed, and if that party broadcasts this message, and every other party broadcasts it again, each party receives the **proposal**. However, this does not work, since before the **proposal** message would arrive at every party, a deputy may be chosen to remove the manager. Another situation could occur that some parties remove the manager, and some follow the **proposal** message. To prevent this situation, the manager first sends the **commit** message. Now if some party receives a **commit** message while a deputy tries to remove the faulty manager, the deputy receives the **proposal** with its **query** message, and then, follows that **proposal** instead of removing the manager. Agreement of the group views is now maintained.

Joining Protocol

This section describes the protocol for a party joining the group.

The main difficulty is to determine the view in which the process is first a member. In the previous protocol, a process p_i installs V_i^{x+1} after receiving a message of the form $\langle commit \{ \langle ready\, p\, q \rangle_{K_j} \}_{p_j \in P} \rangle$ for some $P \subseteq V_i^x$ where $|P| = \lceil (2|V_i^x|+1)/3 \rceil$. For p_i to verify the validity of this message, it must know the content of V^x, because otherwise it is not able to, e.g., determine if P is of the proper size or form. However a joining process may not know the contents of V^x. Thus, the joining protocol must take other measures to ensure that p_i will install a proper V_i^{x+1}.

The basis of the solution is that it suffice for p_i to obtain the contents of some past group views V^y where $y \leq x$, and the **commit** messages sent in views y through x that tell it how to transform V^y into V^{x+1}. To provide those **commit** messages, correct members maintain a **history**[3] set containing, for each prior view, a valid **commit** message sent in that view. Before a correct member installs a new view, it sends its history to the new member $p_{i.}$

For the joiner's part of this protocol, it begins with obtaining some past group view V^y. The joiner then waits to receive a **history** message and, upon receiving one, extracts **commit** messages and construct subsequent views V^z, $z \geq y$. The joining process p_i, continues accepting **history** messages and producing subsequent views to find the first view V^{x+1} that contains its own identifier. It then installs V_i^{x+1} and initiates the normal protocol for that view.

4.5.3 Rampart Protocol

The protocol used in Rampart implements a secure broadcast channel and is composed of several layers: the Atomic multicast protocol, the Reliable multicast protocol and the Echo multicast protocol. These three layers use the *Secure Group Membership Protocol* (Section 4.5.2) to maintain a list of correct parties. This protocol is based under the assumption that at most $\lfloor (|V^x| - 1)/3 \rfloor$ are corrupt.

Multicast semantics

The *Reliable Multicast Protocol* ensures that these predicates hold:

- **Integrity.** For all honest p and m, an honest process executes *R-deliver(p, m)* in view x at most the number of times that p sent *R-mcast(m)* in view x.
- **Uniform Agreement.** If q is an honest member of V^{x+k} for all $k \geq 0$ and an honest p executes *R-deliver(r, m)* in view x, then q executes *R-deliver(r, m)* in view x.
- **Validity-1.** If p is an honest member of V^{x+k} for all $k \geq 0$, then p executes *R-deliver(Vx,.)*
- **Validity-2.** If p and q are honest members of V^{x+k} for all $k \geq 0$ and p executes *R-mcast(m)* in view x, then q executes *R-deliver(p, m)* in view x.

The Atomic Multicast Protocols adds one additional property:

- **Order.** If q is an honest members of V^{x+k} for all $k \geq 0$ and an honest p executes *A-deliver(r, m)* before *A-deliver(r', m')* in view x, then q executes *A-deliver(r, m)* before *A-deliver(r', m')* in view x.

Echo Multicast Protocol

Considered as the core protocol, it ensures that the l-th message received from p for a view x is the same at every honest parties. In the absence of membership changes, a reliable multicast essentially reduces to a single echo multicast.

Using the interface *E-mcast(x, m)*, a party in V^x can multicast a message m to the members of V^x. A process delivers a message m for the view x from p by executing *E-deliver(p, x, m)*.

If a party p wants to send a message to the view V^x, it has to convince every party that it sends the same m everywhere. To do that, it first sends a *init* message that each party will "echoes" by digitally signing m and returning it to p. Once p receives $|P| = \lceil (2|V^x| + 1)/3 \rceil$ echoes for m it sends them in a **commit** message to every members of V^x. Each party has now the assurance that the other received the same m.

3 Do not confuse the **history** message with the *history* of a Web Bulletin Board.

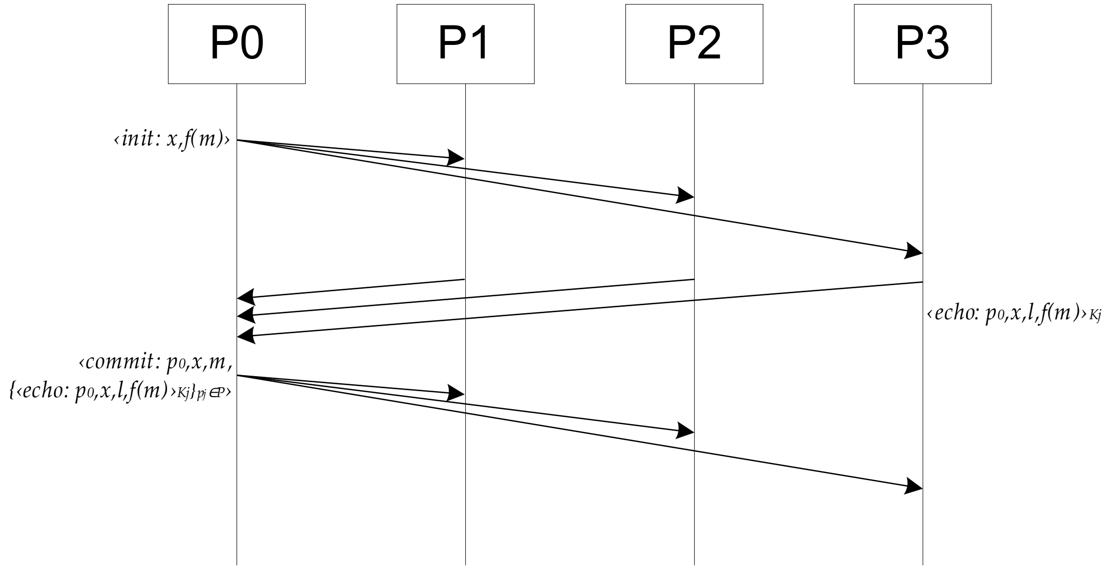

Figure 11: Echo multicast.

The echo multicast protocol then executes as follows. Each $p \in V^x$ maintains a set of counters $\{c_i^x\}_{p_i \in V^x}$, each initially zero, and a set *commits*^{x} of messages, which is initially empty. Each counter c_i^x keeps track of the number of messages that have been E-delivered for view x from p_i, and is used to E-deliver messages from p_i in FIFO order. The steps of the protocol are listed below.

1. If *E-mcast(x, m)* is executed at some $p \in V^x$, p sends $\langle init: x, f(m) \rangle$ to each member of V^x, where f(m) is a hash function.
2. If p_j receives $\langle init: x, d \rangle$ from some $p \in V^x$ and this is the l-th message of the form $\langle \text{init}: \text{x}, ? \rangle$ that p_j has received from p, then p_j sends $\langle echo: p, x, l, d \rangle_{K_j}$ to p.
3. Once the initiator p has received a set of echoes $\{\langle echo: p, x, l, f(m) \rangle_{K_j}\}_{p_j \in P}$ for some l and some $P \subseteq V^x$ where $|P| = \lceil (2|V^x|+1)/3 \rceil$, it sends $\langle commit: p, x, m, \{\langle echo: p, x, l, f(m) \rangle_{K_j}\}_{p_j \in P} \rangle$ to each member of V^x.
4. If a process receives $\langle commit: p_i, x, m, \{\langle echo: p_i, x, l, f(m) \rangle_{K_j}\}_{p_j \in P} \rangle$ for some $l > c_i^x$ and some $P \subseteq V^x$ where $|P| = \lceil (2|V^x|+1)/3 \rceil$, and if it has not received a view V^y, $y > x$, such that $p_i \notin V^y$, then it add this commit message to *commits*^{x}.
5. Whenever a process adds a message $\langle commit: p_i, \ldots \rangle$ to *commits*^{x}, it repeats the following step until it results in no more E-deliveries: if there is a message $\langle commit: p_i, x, m, \{\langle echo: p_i, x, l, f(m) \rangle_{K_j}\}_{p_j \in P} \rangle$ in *commits*^{x} such that $c_i^x + 1 = l$, then it executes *E-deliver(p_i, x, m)* and sets $c_i^x \leftarrow c_i^x + 1$.

After delivering the messages, it stays in the *commits* set until the message is *stable.* A message is stable once every party added it to its commit set. Each party periodically notifies the other parties of the messages in Commits by multicasting a message containing the *c[p]* values. Each party records these values in a set $c_p[p']$. If a party has a message m from p' with index l in its Commits set, and $c_p[p']$ is at least l for each p, the it concludes that m has been added at every party, so it delivers m to a higher protocol layer and removes it from the Commits set. An honest

party does not permit a multicast to remain unstable for longer than a prespecified timeout duration. That is, if a process q retains m in its $commits^x$ beyond some timeout duration after executing *E-deliver*(p_i,x,m), it attempts to make this multicast stable by sending the commit to each party which did not delivered it.

Reliable Multicast Protocol

It ensures that all honest group members deliver the same messages, even in the face of malicious multicast initiators. It consists of two interfaces: *R-mcast(m)* and *R-deliver*(p_i,m).

If the group view does not change, the reliable multicast protocol just relays messages between the echo and the atomic multicast protocol using the *E-mcast(x,m)* and *E-deliver*(p_i,x,m) interfaces of the Echo multicast protocol.

The reliable multicast protocol handles the following tasks : if a group membership change occurs, the new messages must be sent for that new view and messages queue for the old view must be cleaned. Then, the view change may be passed on to higher protocol layers.

When a message m has to be sent, it is echo multicasted in the latest view. The value of the active and closed views are maintained, so that when this message is received, the protocol verifies if the message is accepted. If it is intended for view x, it is delivered and if it is intended for a later view, it is enqueued.

When a group membership change occurs, each party echo multicasts an **end** message to the old view. This message signals that it is the last message it will send in the old view. After receiving an **end** message from every party, each party sends a **flush** message to the new view, containing the Commits set of the echo multicast protocol. This way, agreement is obtained on the messages that have to be delivered in the old view.
No more messages is accepted for the old view. Once a **flush** message is received from every party, the group membership change is announced to the other protocols.
If a party does not send an **end** or **flush** message in time, it is voted out. When a party is voted out during another group membership change, we assume it sent an **end** message and an empty **flush** message.

Atomic Multicast Protocol

It adds the property that honest members deliver these messages in the same order.

It consists of two interfaces: *A-mcast(m)* used to send messages to every parties, and *A-deliver*(p_i,m) is executed on each party when it receives m. The order is assigned using messages broadcasted to the other parties.

It uses the reliable multicast protocol to multicast messages. Once a message m is received from party p it is added to a queue. A designated group member, the *sequencer*, periodically sends an **order** message indicating the order in which the messages are to be delivered.

If a new view is delivered, the atomic multicast protocol does not wait for an order message, but

deterministically chooses an order in which to deliver the queued messages.
If the *sequencer* is corrupted, he might refuse to send **order** messages or tamper with them. So, if an honest party does not A-deliver a message within a predefined timeout period, it requests the sequencer to be removed from the group.

4.5.4 Synchronized Atomic Multicast Protocol

Using the *Atomic Multicast Protocol* ensures that messages are received in the same order, but members that have been offline for a moment do not receive all messages. This protocol enables

parties to synchronize the messages.

To synchronize, the party goes into the *joining* mode and multicasts a message **request-hashes** with the number of messages it received before being voted out of the group. The other parties send the hash of the messages that have been sent after that number. If p receives $\lceil (2|V^x|+1)/3 \rceil$ hashes, whom at least $\lfloor (|V^x|-1)/3 \rfloor +1$ are the same since they come from honest parties. Party p stores the hash of the missing messages requests the missing messages to another party, randomly chosen.
Until p obtained the missing messages, it enqueues any new message to be executed by the *Atomic Multicast Protocol*. After obtaining the missing messages, p delivers them and then delivers the queued messages.

4.5.5 Reading Protocol

When a Reader reads all l messages, the following steps are executed:

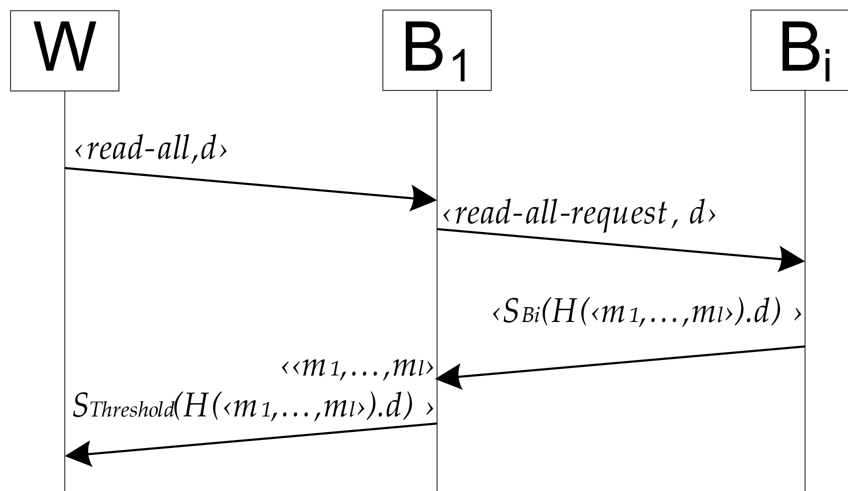

Figure 12: Reading protocol of Peter's Web Bulletin Board.

1. R requests the *history* of B_1 (randomly chosen) and provides a nonce d.
2. B_1 multicasts the message.
3. Each B_i computes a signature share using the messages and the nonce.
4. B_1 combines the shares into a threshold signature and sends it to R.

B_1 multicasts the message to the other B_i using the Synchronized Atomic Multicast protocol (Section 4.5.4).

4.5.6 Writing Protocol

The *Writer* randomly sends his message to B_1, which multicasts it to the group. Figure 13 describes the protocol:

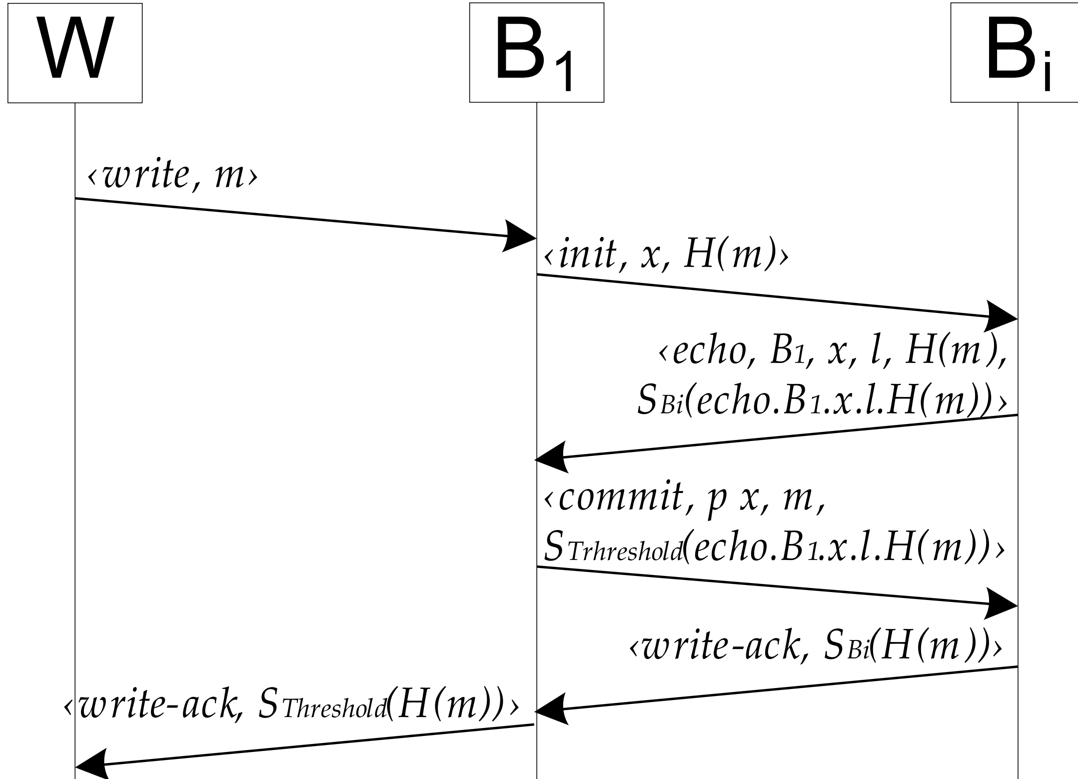

Figure 13: Writing protocol of the Peter's Web Bulletin Board. B_i:all the parties but B_1. x:the view. l:number of message B_i received from B_1.

1. *W* sends his message to B_1.
2. B_1 multicasts the hash of the message to the other boards.
3. Upon receipt, every party sends a signed echo message to B_1.
4. After receiving the echo messages, B_1 combines the signature shares and sends a commit message.
5. After B_i receive the commit message, it publishes it and computes a signature share.
6. B_1 finally combines the signatures shares and returns the receipt to *W*.

If *W* does not receive a receipt after some time, it restarts the protocol with another board than B_1. We make here the assumption that each party checks the signatures, view number, etc.

B_1 multicasts the messages to B_i, using the Synchronized Atomic Multicast protocol (Section 4.5.4).

The difference with the protocol described in [11] is that we use threshold signatures. It does not change the protocol itself but it considerably reduces the amount of data exchanged and makes the signatures easier to test.

4.5.7 Conclusion

The *Web Bulletin Board* described by Peters is based on existing protocols. this point gives more confidence in the system.

The first requirement, availability, is ensured using a distributed *Web Bulletin Board.* No *Single Point of Failure* exists.

Since a receipt is returned to the *Writer*, he can attest that is message hasn't been altered. The

second requirement, unalterable history, is fulfilled.

The third requirement, certified publishing, is only partially respected. Even if the *Writer* signs his message, no time value is given.

Using the *Atomic Multicast Protocol*, the order in which the messages are published is assured.

Group-threshold signatures are used, reducing the amount of data to be transferred and tested.

A weakness of this solution in comparison of the previous candidates is that the *Reader* has less means to test the *history*. He has to trust the system and that at most $\lfloor (n-1)/3 \rfloor$ group members are corrupted.

Since the system is asynchronous and loosely coupled, deadlocks are avoided and timestamps are not used. Therefore we do not need to deal with clock drifts and good performances are provided.

Using the *Secure Group Membership Protocol*, it is possible to add or remove parties and *Rampart Protocol* provides security and a solution to the *Byzantine agreement*.

The implementation can be considered as complicated.

5 Evaluation

A comparison between the different candidates described in Section 4 is done in Table 5 1, using the requirements (Section 2.1) and other interesting properties.

	Availability	Unalterable History	Certified Publishing	Order is respected	Group-Threshold signatures	Reader can test the board	Deadlocks avoided	Byzantine agreement	Possibility to remove and add a party
Trivial Web Bulletin Board	NO	YES	YES	YES	NO	YES	YES	YES	NO
Synchronized Web Bulletin Board	YES	YES	YES	YES	YES	YES	NO	YES	NO
Unsynchronized Web Bulletin Board	YES	YES	YES	NO	NO	YES	YES	NO	NO
Krummenacher's Web Bulletin Board	YES	YES	YES	NO	YES	YES	YES	YES	NO
Peter's Web Bulletin Board	YES	YES	NO	YES	YES	NO	YES	YES	YES

Table 5 1: Comparison of the different candidates.

In Table 5 1, we can see that the availability of the *Trivial Web Bulletin Board* is not ensured. If the *Web Bulletin Board* is victim of a *DOS* attack or suffers from hardware problems, the system stop working. This solution is thus to eliminate.

The *Synchronized Web Bulletin Board* is a possible solution which does not suffers from availability problem, but deadlocks are difficult to eliminate and performances are weak.

In the *Unsynchronized Web Bulletin Board*, the fact that Group-threshold signatures can not be used is a problem. A large amount of data have to be transferred and tested. Also, there is no *Byzantine agreement*, the success of the writing protocol depends of the *Writer*.

The *Krummenacher's Web Bulletin Board* is distributed, it uses Group-threshold signatures and resolves the *Byzantine agreement* problem. The *Reader* is able to test the content of the board. The difficulty of its implement is affordable and good performances are provided. Its major problem is that it does not provide ordering of the messages but depending on what the *Web Bulletin Board* will be used for, it is not important.

Peter's Web Bulletin Board is a distributed solution. Existing protocols like *Rampart* and the *Group Membership Protocol* are used, providing the possibility to add or remove parties and good security mechanisms assuring the consistence of the *history*. However, the *Reader* has less means to test the content of the board than in the previous solutions and certified publishing is not fulfilled. We have to be confident in the fact that less than $\lfloor (n-1)/3 \rfloor$ parties are corrupted. The realization of Peter's Web Bulletin Board can be considered as complicated.

6 Conclusion

In Section 2, we saw that a *Web Bulletin Board* provides the ability to publish something and to detect if its content has been modified.

The requirements of such a system have been described in Section 2. A *Web Bulletin Board* should be available to anyone, no message should modified or moved to another position and for each message we know who posted it.

Five candidate solutions, based on the work of Heather & Lundin and the master thesis of R.A. Peter, have been presented in Section 4. Their architectures, reading and writing protocols are described in detail.

Finally, in Section 5, we saw that two candidates (*Krummenacher's Web Bulletin Board* and *Peter's Web Bulletin Board)* can be considered as the best solutions. Even if they use different mechanisms and are not suitable in every context, they both provide strong security and good performances .

References

[1] Heather, J., Lundin D. The Append-Only Web Bulletin Board, 2009.
[2] R.A. Peters. A Secure Bulletin Board, 2005.
[3] Krantz, Steven G. Zero Knowledge Proofs, 2007.
[4] L. Harn. Group-oriented (t,n) threshold digital signature scheme, 1994.
[5] Alexandra Boldyreva. Efficient threshold signature, multisignature and blind signature scheme based on the gap-diffie-hellman-group signature schemes, 2002.
[6] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract), 1991.
[7] Claus P. Schnorr. Efficient identifications and signatures for smart cards., 1989.
[8] Paul Feldman. A practical Scheme for Non-Interactive Verifiable Secret Sharing, 1987.
[9] Leslie Lamport, Robert E. Shotsak, and Marshall C.Pease. The Byzantine generals problem, 1982.
[10] Roland Krummenacher. Umsetzung eines Web-Bulletin-Boards für E-Voting-Applikationen, 2010.
[11] Michael K. Reiter. Secure Agreement Protocols, 1994.
[12] Michael K. Reiter. A secure group membership protocol, 1996.