Byzantine firing squad using a faulty external source

*Edward T. Ordman**
Memphis State University, Memphis TN 38152 U.S.A.

Abstract

Burns and Lynch have given a general algorithm for initial synchronization of clocks in a computer network, running synchronously, where a limited number of processors are allowed to have *byzantine faults*, that is, fail arbitrarily. They effectively run a byzantine agreement algorithm for each process in the network. Here we show that only one byzantine agreement algorithm is necessary, if we treat the outside signal source as an additional potentially faulty process.

1 Introduction

It is frequently necessary to maintain clocks in each processor in a distributed system, and to attempt to keep these clocks synchronized with one another. A wide variety of abstractions of this problem have been made. Nishitani and Honda [NH] provide a good set of citations for one historical trend, dating back to 1957, in which the processes are viewed as abstract automata, operating synchronously, in a network of initially unknown size and shape, and the goal is to set their clocks to an identical value. This is the *firing squad problem for graphs.* These papers ordinarily assume that the processes and communications links are reliable.

Another abstraction supposes that the processors or communication links may be unreliable, and tries to overcome faults in these components. An extreme form of this is to allow the presence of a limited number of so-called *byzantine* faulty processes, processes that may perform arbitrarily and are not assumed not be locatable by diagnostic tests. Two papers of this type which heavily motivated the present work are [TPS] and [BL].

Burns and Lynch [BL] show that given a fully connected network of n processes, with no more than f faulty and $n > 3f$, and clocks running synchronously but not in initial agreement, the clocks can be synchronized; this is the "byzantine firing squad problem". In [O2] this is extended to a "byzantine firing squad algorithm for graphs" by extending it to a network that is guaranteed to have more than $3f$ processes and to have more than $2f$ disjoint routes between each pair of processes, but where the processes do not initially know the total number of processes or have a map of the graph beyond their immediate neighbors.

The Burns and Lynch algorithm supposes that at a given time each process may or may not receive a **START** signal from "the outside"; the goal is to set a time when all processes can

*Partially supported by U.S. National Science Foundation grant number DCR-8503922.

agree on whether a sufficient number received such a **START** signal. Their solution, briefly, is to have each process that receives a **START** signal notify all the other processes of this fact using a byzantine agreement algorithm. Their method will work with any byzantine agreement algorithm with a few standard properties. At the end of a known time after the initial notification, either all correct processes *accept* the message or none do. So at certain given times, any correct process knows how many notifications of **START** messages have been accepted by all correct processes, and can use that number as a decision variable (e.g., reset my clock to zero if that number has just exceeded $f + 1$ for the first time). This method requires a byzantine agreement algorithm on a vector, one value for each processor in the system, or equivalently, a number of byzantine agreement algorithms equal to the number of processes (the latter approach is required in [O2], where the number of processes is initially unknown).

In the present paper we show that only one byzantine agreement process (on one bit) is necessary, provided that it is slightly modified to treat the "outside world" as an additional (potentially faulty) process. While this means there are potentially $f + 1$ faulty processes, this turns out to be permissible since the extra one does not participate in the entire algorithm, just the initial part; it does lengthen the byzantine agreement algorithm somewhat since the algorithm must have length in rounds exceeding the number of faulty processes (see [FLM] for an easy proof), apparently including the faulty additional "outside" process.

For concreteness we use a byzantine agreement algorithm simplified from that of [O2], an adaptation of that in [TPS]. The broadcast, byzantine agreement, and firing squad algorithms of [O2] are designed to work in a network which is not fully connected and where common clock settings are not available at the outset. The simplification we use here is for a completely connected network like that of [BL] or [TPS], but phrased so that a reading of this paper together with [O2] will show how to extend the result to a more general network such as that of [O2].

2 Description of environment

We are given a collection of processes $P_i, i = 1, \ldots, n$. We assume that each pair of processes is connected by a two-way communications channel. We will refer to process P_i as "process i."

We assume that at least $n - f$ of the processes are correct, that is, follow the algorithms we give. The other processes (at most f) are faulty and we do not attempt to control their behaviour. "Correct process i ..." means "Process P_i, assuming it is correct ...."

We assume that $n > 3f$; see [FLM] for an easy proof that this is needed.

We assume that each correct process P_i has the following available to it at all times:

(i) its identifying number i and the number n.

(ii) the ability to tell from which of its neighbors it received a message (hence, a faulty process cannot lie to its neighbors about its own identity.)

(iii) The maximum number f of faulty processes. (An algorithm designed to defend against no more than f faulty processes is called an *f-resilient algorithm*).

We assume a synchronous system in which computation proceeds in rounds. In each round, each correct process does in order the following:

(i) it receives an arbitrary number of messages from other processes;

(ii) it performs some computation based on its prior state and the messages just received;

(iii) it sends an arbitrary number of messages to other processes; these messages will be received at the start of the next round.

We assume each correct process i has an internal clock which advances once per round; during any given round process i knows its "internal time" t_i and this is constant throughout a round.

We do not assume, of course, that the various clocks agree. While we put few restraints on faulty processes, we assume that they cannot send so many messages as to overwhelm the correct processes. We suppose that faulty processes may otherwise act in an arbitrary manner. In particular they may fail to forward messages, alter them, or invent them; they may break cryptographic codes, forge codes, forge signatures; "wiretap" messages between other pairs of processes; act cooperatively; pass diagnostic tests but malfunction otherwise; act exactly correctly throughout or until some critical moment. On the other hand, our upper bound of f faulty processes is global: at most f processes can be faulty during any part of the algorithms here described, and a process is counted as a faulty process whether it violates the algorithm in a trivial single event or is malfunctioning throughout.

We assume that each correct process has an input port from "the outside world" through which we can provide a message (e.g. a proposed clock setting, or simply a message asking the processes to synchronize their clocks).

For simplicity in the arguments, we will assume the existence of an external clock C (not visible to the correct processes) that advances when their clocks do. In addition, we assume that we are capable of sending messages from the outside world to the correct processes so that these inputs will arrive in the same round or in a desired set of rounds.

3 A broadcast algorithm for an internal source

When correct process i wishes to send a message, it sends to a neighbor a text string, which may be arbitrarily encoded. E.g. the text may be a construction containing some basic information ("I want to synchronize clocks") plus a number of data items essential to our algorithms, which we will denote by boldface **KEYWORDS**. Whenever such a string passes from process i to process j we say that process i *sends* it and process j *receives* it.

A process is said to *hear* a message if one of the following conditions is met:

(i) it sent it itself in the previous round; or

(ii) it received the message at the start of this round.

Our notation below carefully distinguishes *receiving* and *hearing* a message to enable generalization; in the case of a distributed (not completely connected) network, *hearing* a message is a more complicated process since it may require receiving copies of the message that have been forwarded by at least $f + 1$ disjoint routes [O2].

For timing consistency, we may suppose each process sending a message to the others also sends a copy to itself, receiving it the following round.

We require a way for a correct process to get a message to all correct processes, and to be confident others have gotten the same message it has. We call the algorithm we need a *timed broadcast algorithm.*

For a process i to *broadcast* a message, it participates in the following *timed broadcast algorithm.* It sends the message (**INIT** *"text"*) to all other processes. Each correct process j stands ready at all times to do the following:

It saves all messages heard noting in which round (by its local clock) they were received. When it hears a message of the form (**INIT** *"text"*) from process i, it sends (**ECHO** *"text"* **at now** $-$ 1 **from** i) to all other processes. This shows both the originator i of the message being echoed and the elapsed time (1) since the message was originally sent.

Process j also *echos* the message (sends such an **ECHO** message) if it hears that the same message has been **ECHO**ed by $f+1$ distinct processes. That is, if it hears no **INIT** of a particular message from a particular process i, but it hears $f+1$ **ECHO**s reporting the same text was originated at the same time (by j's clock) by the same process i, where these $f+1$ **ECHO**s come from distinct processes, then process j itself sends such an **ECHO** to all the processes with an "... **at now** $-$ e ..." clause telling how long ago the message **INIT** occurred.

Finally, if a process hears the same message (including the same time of origin) **ECHO**ed by $2f+1$ distinct processes, it *accepts* the message.

Now if a correct process broadcasts a message, all correct processes hear the **INIT** in the next round and send an **ECHO**; after the next round every correct process receives at least $n-f$ (at least $2f+1$) **ECHO**s and accepts the message.

It is now easy to confirm the following properties (proofs are omitted but are easy and are very similar to those in [TPS]):

Correctness. If correct process i broadcasts a message in round t (on the external clock), all correct processes accept it by round $t+2$.

Unforgeability. If correct process j accepts a message from process i and process i is correct, then i did broadcast that message at the time believed by j.

Relay. If a correct process j accepts a message from a (correct or incorrect) process i in round t (external clock), then all correct processes accept this message (and the same originator and time of sending) no later than round $t+2$.

4 The Byzantine Agreement algorithm

Suppose a process j (not necessarily correct) may have broadcast a message "text" at time t. We need an algorithm that will let all correct processes settle on a time when they will all know whether or not j broadcast that message. In other words, the danger in the broadcast "acceptance" algorithm above is that no matter how long process i waits without accepting the message from j, it can't stop and be sure that some other message hasn't accepted it during the last round. If it accepts a message and takes some action based on it, it can't be sure that some other correct process will not wait a round or two later before acting. Our cure for this problem is a variation of the byzantine agreement algorithm of [TPS].

Each correct process j acts as follows: if it sent a message (**INIT** *"text"*) at time t_j (on its local clock), it simultaneously decides to agree that this message was sent and broadcasts during the same round a message (**INIT** j **agrees** j **sent** *"text"* **at now** $-$ 0). (In fact, since every sender of the simple **INIT** must send this, the first message can be interpreted by all to include the second, and the second needn't be physically sent). Every other correct process i treats j's original (**INIT** *"text"*) as follows:

BYZANTINE AGREEMENT ALGORITHM:

If, for some p (from 1 to $f+1$ inclusive),

(a) i accepts at least p messages of the form (... k **agrees** j **sent** *"text"* **at now** $-$ e) from p distinct processes k, all agreeing on the "text" and the time j sent "text" (as computed by i using the various elapsed times e); and

(b) the message ... j **agrees** j **sent** ... was broadcast in the same round j reportedly sent "text"; and

(c) if p is greater than 1, at least one of the ... k **agrees** j **sent** ... messages was broadcast at each of the rounds $2, 4, \ldots, (2p-2)$ after the initial message from j;

then process i *decides to agree* at that time and during round $2p$ after the initial message process i broadcasts a message of the form (**INIT** i **agrees** j **sent** *"text"* **at now** $- 2p$).

By a proof like that in [TPS] we now have:

Theorem 4.1 *If any correct process decides to agree a message was sent, then they all decide to agree no later than* $2(f+1)$ *rounds after the message was sent.* □

Now, if a process has *decided to agree* that j **sent** *"text"* at a given time by $2(f+1)$ after the given time, we say it actually *agrees* to that message during round $2(f+1)$ after the given time. (We distinguish *decides to agree* from *agrees*). This form of *byzantine agreement* has the following nice properties:

Agreement. Either all correct processes agree that j sent "text" in a given prior round, or no correct process agrees.

Timed Decision. If all correct processes agree that j sent "text" in a given round, then all of them first agree to that in the same round, $2(f+1)$ rounds after it was sent.

Validity. If j is a correct process and broadcast "text", all correct processes agree that it did so; if j is correct and did not broadcast "text", no correct process agrees that it did so.

5 A basic Firing Squad algorithm

In this section we restate and specialize a (much more general) firing squad algorithm due to [BL]. Suppose that in a given round t (as viewed by the outside clock) we send a signal **START** from the outside to some (possibly none) of the processes. Each correct process j receiving a **START** from the outside world broadcasts (**INIT START**) and begins a byzantine agreement algorithm. If any correct process receives a **START**, all correct processes will *decide to agree* to that fact and will *agree* to it in round $t + 2(f+1)$. Accordingly, in any round in which a correct process agrees to any **START** message, it should check and see how many distinct **START** messages it is agreeing to at this time. For instance, consider what we will call "Algorithm 1":

ALGORITHM 1: following the first round in which a process agrees to at least $f+1$ **START** messages, it sets its clock to 0 for the start of the next round. □

This is a restatement of the algorithm of Burns and Lynch, for this particular byzantine agreement algorithm; they state it more generally and carefully compute the complexity (number of bits of messages required).

Since in any given round all correct processes agree to anything any one of them does, we immediately see that *Algorithm 1* has the following properties:

Time Agreement. If any correct process sets its clock to 0 as a result of *Algorithm 1*, all other correct processes also do so in the same round.

Safety. No correct process sets its clock to 0 as a result of *Algorithm 1* unless at least 1 correct process received a **START** signal from outside. (For if none did, at most the f faulty processes broadcast **START**).

Liveness. If $2f+1$ processes received **START** signals from outside at the start of round t, then all correct processes set their clocks to 0 at the start of round $t + 2(f+1) + 1$. (For at least $f+1$ correct processes received and broadcast a **START** signal).

Burns and Lynch point out that variations are possible. For example, they classify the above as a *Strict* algorithm because of the **Safety** property. For a *Permissive* algorithm, we could abandon the need for **Safety** and synchronize on the agreement to any one **START** signal (so that sending $f+1$ from outside would guarantee synchronization).

6 Using an outside participant

We now show how to use just one (slightly modified) byzantine agreement algorithm instead of requiring one for each process (or at least each **START** signal, as *Algorithm 1* did. Instead of trying to reach agreement on a number of messages (... j **sent START at now** $- e$) we can reach agreement on the single message (**OUTSIDE sent START at now** $- e$). Suppose each process receiving **START** treats it as (**INIT START**) received from a process numbered zero (the **OUTSIDE** process), and immediately echos it. It also supposes receipt of the usual (**INIT** 0 **agrees** 0 **sent** ...) message. All algorithms proceed as usual; in counting $f + 1$ or $2f + 1$ copies of messages the ones from **OUTSIDE** are not counted. The proofs that the *broadcast* and *byzantine agreement* algorithms work go through as usual (the **OUTSIDE** may be correct or faulty) with one exception: in the *byzantine agreement* algorithm and Theorem 4.1, there are now potentially $f + 1$ faulty processes counting the **OUTSIDE**. We will see that agreement cannot be relied on before round $(2f + 2) + 1$ after the message is sent, and synchronization occurs in round $(2f + 2) + 2$ after the **START** signals are sent from **OUTSIDE**.

We need to provide a modification of the *byzantine agreement algorithm* for the special case when the broadcaster is **OUTSIDE**. Any correct process receiving an appropriate signal (such as **START**) on its line from outside the network will treat this as if it is a correctly formatted message of the form (**INIT START**) from process 0. It will also act as if this were followed the next time by a correctly formatted **ECHO** from process 0. That is, each correct process receiving a **START** acts as if it came from a correct neighbor; each correct process not receiving a **START** acts as if it had a neighbor with identifier zero that was dead and nonresponsive (hence faulty). In fact, this produces the desired result for a "strict" firing squad algorithm, in a sense that will shortly become clear. We summarize the following arguments adapted from [TPS]:

Lemma 6.1 *If at least* $2f + 1$ **START** *signals are sent from the outside in a given round, all correct processes will accept a* **START** *signal from process* 0 *after* 3 *rounds. If no* **START** *signals are sent in a given round, no correct process will accept a* **START** *signal* 3 *rounds later.*

PROOF: If $2f + 1$ **START** signals are sent from outside in round t (by the external clock), then at least $f + 1$ are received by correct processes; so these $f + 1$ processes **ECHO** the message in round $t + 1$. Thus all correct processes hear $f + 1$ **ECHO**s and issue their own **ECHO** by round $t + 2$, and all correct processes accept the message by round $t + 3$. If no **START** signal is sent from outside, no correct process receives one. At most f processes, all faulty, can send **ECHO**s, so no correct process accepts the message. □

It is now clear how to modify the timed byzantine agreement algorithm. The correct processes must decide if they agree that (... 0 **sent START**...). Again, each process receiving a **START** from outside acts as if it simultaneously received from process 0 a message (**INIT** 0 **agrees** 0 **sent START at now** $-$ 0). It is this message, rather than the simple **START**, which is actually forwarded for potential agreement. The process is now exactly as in the *byzantine agreement algorithm* of Section 4, but with $p = 1$ to $f + 2$ instead of $p = 1$ to $f + 1$, decisions to agree taken at times $2+1$, $4+1$, $6+1$, and so on, and with the final agreeing taking place $2(f+2)+1$ rounds following the given round. The offset of 1 is because accepting the first message from outside may take 3 rounds instead of 2 (the outside world is to have its message accepted even if it sends **START** to just $f + 1$ correct processes instead of to all correct processes). The need for $2(f + 2)$ instead of $2(f + 1)$ follows from the addition of the new possibly faulty "outside" process. This will be clear if we restate part of the proof that the byzantine agreement algorithm works.

Lemma 6.2 *If no correct process decides to agree that (...* 0 **sent START** ... *) by time* $2(f+1)+1$ *after* **START** *was allegedly sent, no correct process will subsequently decide to agree.*

PROOF: A correct process can *decide to agree* only if it accepts at least one (... k **agrees** 0 **sent START at now** $-$ ...) broadcast at each of the times $0, 2+1, 4+1, \ldots$ up to the time it decides. If the present is past $2(f+1)+1$ (relative to the initial message) this requires that $f+2$ distinct processes have broadcast (... k **agrees** ...) messages. But there are at most f faulty processes plus the outside process: so at most $f+1$ can have done so if no correct process has. Hence, a correct process arriving at time $2(f+2)+1$ without deciding to agree will never decide to agree. □

Lemma 6.3 *If any correct process* decides to agree *that "...*0 **sent START**... *" no later than* $2(f+1)+1$ *rounds after it was sent, then every correct process* decides to agree *by round* $2(f+2)+1$ *after it was sent.*

PROOF: If any correct process j *decides to agree*, say in round $2s+1$, this means it has *accepted* a (...0 **agrees** 0 **sent START**...) message which was broadcast at round 0 relative to the **START**, and has *accepted* at least $s-1$ other (... j **agrees** 0 **sent START**...) messages, one broadcast at each of the times $2+1, 4+1, \ldots, 2(s-1)+1$. By the **Relay** property of the broadcast algorithm, every correct process will have accepted all of these by time $2(s+1)+1$; by the **Correctness** property, all correct processes accept the message (... j **agrees** 0 **sent START** ...), broadcast in round $2s+1$, by round $2(s+1)+1$; this is enough to make all correct processes decide to agree in that round. □

Thus we have achieved a "strict" byzantine firing squad algorithm needing approximately as many messages as a single byzantine agreement algorithm, rather than approximately n times as many.

7 A permissive algorithm and other modifications

The above method works well for a strict algorithm and for the case when we only worry about counting messages resulting from actual **START** messages from "outside" in a single round. [BL] also treat the questions of messages resulting from false "START" messages originated by the faulty processes and permissive algorithms.

In the permissive case, we want the sending of $f+1$ **START** messages from outside to guarantee synchronization. In exchange, we must accept the fact that the faulty processes can force a synchronization by themselves. Again, the modification is easy. If $f+1$ **START** signals are sent from outside, at least one correct process j will receive one. That process can broadcast (... j **agrees** 0 **sent START at now** -1) Any correct process accepting any message of that form can translate the first such message it accepts to include the message (...0 **agrees** 0 **sent START at** [the same time]). Thus every correct message will promptly decide to agree and synchronization will occur as desired. Of course, any faulty process could do the same thing at any time and force synchronization.

Permitting **START** signals to arrive at different times is also easy. Suppose we wish to have a *strict* algorithm in which the **START** signals can arrive over a period of time. That is, synchronization is permitted if any one **START** signal arrives from outside and may occur $2(f+2)+1$ or more rounds after any signal is received; if at least $2f+1$ **START** signals are sent from outside, synchronization must occur within $2(f+2)+1$ rounds after signal number $2f+1$. (This flexibility is required since the synchronization will occur $2(f+2)+1$ rounds after the $f+1$–st process announces it has received a **START**; but that process may be faulty, and up to f of the **START** signals may go to faulty processes.) To achieve this, we simply "latch" **START** signals. That is, each correct process receiving a **START** from outside acts as if it receives a **START** from outside in each round thereafter. As soon as $f+1$ processes report receiving a

START in the same round (which cannot happen until one correct process receives a **START**, and must happen once $2f + 1$ processes receive **START**s), it is guaranteed that synchronization will occur $2(f + 2) + 1$ rounds later.

The strategy of the previous paragraph would be most inefficient if it actually required repeating all the needed messages in each round. Luckily, we need not. Any process accepting any **START** as having been sent at time $(\ldots \mathbf{now} - x \ldots)$ will also accept that it has been sent "again" at each time following that. Hence any message relevant to accepting or agreeing to a **START** need in fact be sent only once; all processes can regard it as being received repeatedly in each round thereafter.

Finally, can we reduce the message load induced by spurious messages started by the faulty processes? Doing this thoroughly requires further analysis: how soon can correct processes recognize faulty processes and begin to ignore messages from them? This appears to be an interesting question even in a fully connected network and extremely difficult in a truly distributed network. The method of [TPS] causes a byzantine agreement algorithm to stop earlier than the form here if several faulty processes behave themselves for part of the algorithm. That is, the process stops after $2(k + 2)d + 1$ rounds unless more than k processes malfunction in a particular way during the algorithm. This feature of the [TPS] algorithm has been eliminated in the version here for brevity, but is easy to restore. Note that we could not actually use the early-stopping in the present application, since we needed a fixed stopping time for the algorithm. Different correct processes may observe different numbers of faulty processes (a faulty process may appear correct to one correct process and faulty to another) and may decide whether or not to agree at different times in the [TPS] version. We need to postpone the agreement to the correct time, for the firing squad result to hold; but the earlier "decision time" in the [TPS] algorithm is a time after which no further messages need be forwarded if the process has decided not to agree. (If it has decided to agree, it must forward messages necessary to helping other correct processes agree).

Acknowledgement

This work would not have been possible without several extremely helpful conversations with James E. Burns.

References

[BL] J.E. Burns and N.A. Lynch, *The byzantine firing squad problem*, MIT Laboratory for Computer Science report TM-275, April 1985; *to appear in* Advances in Computing Research.

[FLM] M.J. Fischer, N.A. Lynch and M. Merritt, *Easy impossibility proofs for distributed consensus problems*, Proc. 4th ACM Symp. on Principles of Distributed Computing, Minaki, Canada, August, 1985, 59-70.

[NH] Y. Nishitani and N. Honda, *The firing squad synchronization problem for graphs*, Theoretical Computer Science 14(1981), 39-61.

[O] E.T. Ordman, *Distributed graph recognition with malicious faults*, to appear in Proceedings First China-U.S.A. International Conf. on Graph Theory and its Applications, Jinan, China, June, 1986.

[O2] E.T. Ordman, *A byzantine firing squad algorithm for networks*, Technical report, Memphis State University Dept. of Mathematical Sciences, 1987.

[TPS] S. Toueg, K.J. Perry and T.K. Srikanth, *Fast distributed agreement*, Proc. 4th ACM Symposium on Principles of Distributed Computing, Minaki, Canada, August, 1985, 87-101.