Chapter 10

Byzantine agreement

Like synchronous agreement (as in Chapter 9) except that we replace crash failures with **Byzantine failures**, where a faulty process can ignore its programming and send any messages it likes. Since we are operating under a universal quantifier, this includes the case where the Byzantine processes appear to be colluding with each other under the control of a centralized adversary.

10.1 Lower bounds

We'll start by looking at lower bounds.

10.1.1 Minimum number of rounds

We've already seen an $f+1$ lower bound on rounds for crash failures (see §9.3). This lower bound applies *a fortiori* to Byzantine failures, since Byzantine failures can simulate crash failures.

10.1.2 Minimum number of processes

We can also show that we need $n > 3f$ processes. For $n = 3$ and $f = 1$ the intuition is that Byzantine B can play non-faulty A and C off against each other, telling A that C is Byzantine and C that A is Byzantine. Since A is telling C the same thing about B that B is saying about A, C can't tell the difference and doesn't know who to believe. Unfortunately, this tragic soap opera is not a real proof, since we haven't actually shown that B can say exactly the right thing to keep A and C from guessing that B is evil.

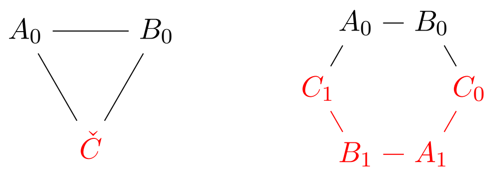

Figure 10.1: Three-process vs. six-process execution in Byzantine agreement lower bound. Processes A_0 and B_0 in right-hand execution receive same messages as in left-hand three-process execution with Byzantine $\check{C}$ simulation C_0 through C_1. So validity forces them to decide 0. A similar argument using Byzantine $\check{A}$ shows the same for C_0.

The real proof:[1] Consider an artificial execution where (non-Byzantine) A, B, and C are duplicated and then placed in a ring $A_0B_0C_0A_1B_1C_1$, where the digits indicate inputs. We'll still keep the same code for $n = 3$ on A_0, B_0, etc., but when A_0 tries to send a message to what it thinks of as just C we'll send it to C_1 while messages from B_0 will instead go to C_0. For any adjacent pair of processes (e.g. A_0 and B_0), the behavior of the rest of the ring could be simulated by a single Byzantine process (e.g. C), so each process in the 6-process ring behaves just as it does in some 3-process execution with 1 Byzantine process. It follows that all of the processes terminate and decide in the unholy 6-process Frankenexecution[2] the same value that they would in the corresponding 3-process Byzantine execution. So what do they decide?

Given two processes with the same input, say, A_0 and B_0, the giant execution is indistinguishable from an $A_0B_0\check{C}$ execution where $\check{C}$ is Byzantine (see Figure 10.1. Validity says A_0 and B_0 must both decide 0. Since this works for any pair of processes with the same input, we have each process deciding its input. But now consider the execution of $C_0A_1\check{B}$, where $\check{B}$ is Byzantine. In the big execution, we just proved that C_0 decides 0 and A_1 decides 1, but since the C_0A_1B execution is indistinguishable from the big execution to C_0 and A_1, they do the same thing here and violate agreement.

This shows that with $n = 3$ and $f = 1$, we can't win. We can generalize this to $n = 3f$. Suppose that there were an algorithm that solved Byzantine agreement with $n = 3f$ processes. Group the processes into groups of size f, and let each of the $n = 3$ processes simulate one group, with everybody in

[1] The presentation here is based on [AW04, §5.2.3]. The original impossibility result is due to Pease, Shostak, and Lamport [PSL80]. This particular proof is due to Fischer, Lynch, and Merritt [FLM86].

[2] Not a real word.

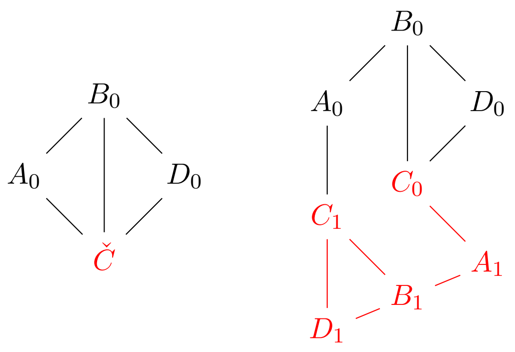

Figure 10.2: Four-process vs. eight-process execution in Byzantine agreement connectivity lower bound. Because Byzantine $\check{C}$ can simulate C_0, D_1, B_1, A_1, and C_1, good processes A_0, B_0 and D_0 must all decide 0 or risk violating validity.

the group getting the same input, which can only make things easier. Then we get a protocol for $n = 3$ and $f = 1$, an impossibility.

10.1.3 Minimum connectivity

So far, we've been assuming a complete communication graph. If the graph is not complete, we may not be able to tolerate as many failures. In particular, we need the connectivity of the graph (minimum number of nodes that must be removed to split it into two components) to be at least $2f+1$. See [Lyn96, §6.5] for the full proof. The essential idea is that if we have an arbitrary graph with a vertex cut of size $k < 2f+1$, we can simulate it on a 4-process graph where A is connected to B and C (but not D), B and C are connected to each other, and D is connected only to B and C. Here B and C each simulate half the processes in the size-k cut, A simulates all the processes on one side of the cut and D all the processes on the other side. We then construct an 8-process artificial execution with two non-faulty copies of each of A, B, C, and D and argue that if one of B or C can be Byzantine then the 8-process execution is indistinguishable to the remaining processes from a normal 4-process execution. (See Figure 10.1.)

An argument similar to the $n > 3f$ proof then shows we violate one of validity or agreement: if we replacing C_0, C_1, and all the nodes on one side of the $C_0 + C_1$ cut with a single Byzantine $\check{C}$, we force the remaining non-faulty nodes to decide their inputs or violate validity. But then doing the same thing with B_0 and B_1 yields an execution that violates agreement.

Conversely, if we have connectivity $2f+1$, then the processes can simulate a general graph by sending each other messages along $2f+1$ predetermined vertex-disjoint paths and taking the majority value as the correct message. Since the f Byzantine processes can only corrupt one path each (assuming the non-faulty processes are careful about who they forward messages from), we get at least $f+1$ good copies overwhelming the f bad copies. This reduces the problem on a general graph with sufficiently high connectivity to the problem on a complete graph, allowing Byzantine agreement to be solved if the other lower bounds are met.

10.1.4 Weak Byzantine agreement

(Here we are following [Lyn96, §6.6]. The original result is due to Lamport [Lam83].)

Weak Byzantine agreement is like regular Byzantine agreement, but validity is only required to hold if there are no faulty processes at all.[3] If there is a single faulty process, the non-faulty processes can output any value regardless of their inputs (as long as they agree on it). Sadly, this weakening doesn't improve things much: even weak Byzantine agreement can be solved only if $n \geq 3f+1$.

Proof: As in the strong Byzantine agreement case, we'll construct a many-process Frankenexecution to figure out a strategy for a single Byzantine process in a 3-process execution. The difference is that now the number of processes in our synthetic execution is much larger, since we want to build an execution where at least some of our test subjects think they are in a non-Byzantine environment. The trick is to build a very big, highly-symmetric ring so that at least some of the processes are so far away from the few points of asymmetry that might clue them in to their odd condition that the protocol terminates before they notice.

Fix some protocol that allegedly solves weak Byzantine agreement, and let r be the number of rounds for the protocol. Construct a ring of $6r$ processes $A_{01}B_{01}C_{01}A_{02}B_{02}C_{02}\dots A_{0r}B_{0r}C_{0r}A_{10}B_{10}C_{10}\dots A_{1r}B_{1r}C_{1r}$, where each X_{ij} runs the code for process X in the 3-process protocol with input i. For each adjacent pair of processes, there is a 3-process Byzantine

[3]An alternative might be to weaken agreement or termination to apply only if there are no non-faulty processes, but this makes the problem trivial. If we weaken agreement, we can just have each process decide whatever process 1 tells it to, and if we weaken termination, we can do more or less the same thing except that we only terminate if all the other processes tell us they heard the same value from process 1.

execution which is indistinguishable from the $6r$-process execution for that pair: since agreement holds in all Byzantine executions, each adjacent pair decides the same value in the big execution and so either everybody decides 0 or everybody decides 1 in the big execution.

Now we'll show that means that validity is violated in some no-failures 3-process execution. We'll extract this execution by looking at the execution of processes $A0_{r/2}B0_{r/2}C0_{r/2}$. The argument is that up to round r, any input-0 process that is at least r steps in the ring away from the nearest 1-input process acts like the corresponding process in the all-0 no-failures 3-process execution. Since $A_{0,r/2}$ is $3r/2 > r$ hops away from A_{1r} and similarly for $C_{0,r/2}$, our 3 stooges all decide 0 by validity. But now repeat the same argument for $A_{1,r/2}B_{1,r/2}C_{1,r/2}$ and get 3 new stooges that all decide 1. This means that somewhere in between we have two adjacent processes where one decides 0 and one decides 1, violating agreement in the corresponding 3-process execution where the rest of the ring is replaced by a single Byzantine process. This concludes the proof.

This result is a little surprising: we might expect that weak Byzantine agreement could be solved by allowing a process to return a default value if it notices anything that might hint at a fault somewhere. But this would allow a Byzantine process to create disagreement revealing its bad behavior to just one other process in the very last round of an execution otherwise headed for agreement on the non-default value. The chosen victim decides the default value, but since it's the last round, nobody else finds out. Even if the algorithm is doing something more sophisticated, examining the $6r$-process execution will tell the Byzantine process exactly when and how to start acting badly.

10.2 Upper bounds

Here we describe two upper bounds for Byzantine agreement, one of which gets an optimal number of rounds at the cost of many large messages, and the other of which gets smaller messages at the cost of more rounds. (We are following §§5.2.4–5.2.5 of [AW04] in choosing these algorithms.) Neither of these algorithms is state-of-the-art, but they demonstrate some of the issues in solving Byzantine agreement without the sometimes-complicated optimizations needed to get all the parameters of the algorithm down simultaneously.

10.2.1 Exponential information gathering gets $n = 3f + 1$

The idea of **exponential information gathering** is that each process will do a lot of gossiping, but now its state is no longer just a flat set of inputs, but a tree describing who it heard what from. We build this tree out of pairs of the form (path, input) where path is a sequence of intermediaries with no repetitions and input is some input. A process's state at each round is just a set of such pairs. At the end of $f + 1$ rounds of communication (necessary because of the lower bound for crash failures), each non-faulty process attempts to untangle the complex web of hearsay and second-hand lies to compute the same decision value as the other processes. This technique was used by Pease, Shostak, and Lamport [PSL80] to show that their impossibility result is tight: there exists an algorithm for Byzantine agreement that runs in $f + 1$ synchronous rounds and guarantees agreement and validity as long as $n \geq 3f + 1$.

```
$S \leftarrow \{\langle\langle\rangle, \mathsf{input}\rangle\}$
for round $\leftarrow 0 \dots f$ do
    Send $\{\langle xi, v\rangle \mid \langle x, v\rangle \in S \wedge |x| = \mathsf{round} \wedge i \notin x\}$ to all processes
    upon receiving $S'$ from $j$ do
        // Filter out obviously bogus tuples
        if $\forall \langle xj', v\rangle \in S' : |x| = \mathsf{round} \wedge j' = j$ then
            $S \leftarrow S \cup S'$

// Compute decision value
for each path $w$ of length $f + 1$ with no repeats do
    if $\langle w, v\rangle \in S$ for some $v$ then
        Let $\mathsf{val}'(w, i) = v$
    else
        Let $\mathsf{val}'(w, i) = 0$
for each path $w$ of length $f$ or less with no repeats do
    Let $\mathsf{val}'(w, i) = \mathrm{majority}_{j \notin w} \mathsf{val}(wj, i)$
Decide $\mathsf{val}'(\langle\rangle, i)$
```

Algorithm 10.1: Exponential information gathering. Code for process i.

The algorithm is given in Algorithm 10.1. The communication phase is just gossiping, where each process starts with its only its input and forwards any values it hears about along with their provenance to all of the other processes. At the end of this phase, each process has a set of pairs of the form $(\mathsf{path}, \mathsf{value})$ where path spans all sequences of 0 to $f+1$ distinct ids and value is the input value forwarded along that path.

We write $\mathsf{val}(w,i)$ for the value stored in i's list at the end of the protocol that is associated with path w. Because we can't trust these $\mathsf{val}(w,i)$ values to be an accurate description of any process's input if there is a Byzantine process in w, each process computes for itself replacement values $\mathsf{val}'(w,i)$ that use majority voting to try to get a more trustworthy picture of the original inputs.

Formally, we think of the set of paths as a tree where w is the parent of wj for each path w and each id j not in w. To apply EIG in the Byzantine model, ill-formed messages received from j are treated as missing messages, but otherwise the data-collecting part of EIG proceeds as in the crash failure model. However, we compute the decision value from the last-round values recursively as follows. First replace any missing pair involving a path w with $|w| = f+1$ with $(w,0)$. Then for each path w, define $\mathsf{val}'(w,i)$ to be the majority value among $\mathsf{val}'(wj,i)$ for all j, or $\mathsf{val}(w,i)$ if $|w| = f+1$. Finally, have process i decide $\mathsf{val}'(\langle\rangle, i)$, which it can compute locally from its own stored values $\mathsf{val}(w,i)$.

Each val' is a reconstruction of older values from later ones. As we move up the tree from wj to w we are moving backwards in time, until in the end we get the decision value $\mathsf{val}'(\langle\rangle, i)$ as a majority of reconstructed inputs $\mathsf{val}'(j,i)$. One way to think about this is that I don't trust j to give me the right value for wj—even when $w = \langle\rangle$ and j is claiming to report its own input—so instead I take a majority of values of wj that j allegedly reported to other people. But since I don't trust those other people either, I use the same process recursively to construct those reports.

10.2.1.1 Proof of correctness

This is just a sketch of the proof from [Lyn96, §6.3.2]; essentially the same argument appears in [AW04, §5.2.4].

We start with a basic observation that good processes send and record values correctly:

Lemma 10.2.1. *If i, j, and k are all non-faulty then for all w,* $\mathsf{val}(wk,i) = \mathsf{val}(wk,j) = \mathsf{val}(w,k)$.

Proof. Trivial: k announces the same value $\mathsf{val}(w,k)$ to both i and j. □

More involved is this lemma, which says that when we reconstruct a value for a trustworthy process at some level, we get the same value that it sent us. In particular this will be used to show that the reconstructed inputs $\mathsf{val}'(j, i)$ are all equal to the real inputs for good processes.

Lemma 10.2.2. *If j is non-faulty then $\mathsf{val}'(wj, i) = \mathsf{val}(w, j)$ for all non-faulty i and all w.*

Proof. By induction on $f + 1 - |wj|$. If $|wj| = f + 1$, then $\mathsf{val}'(wj, i) = \mathsf{val}(wj, i) = \mathsf{val}(w, j)$. If $|wj| < f + 1$, then $\mathsf{val}(wj, k) = \mathsf{val}(w, j)$ for all non-faulty k. It follows that $\mathsf{val}(wjk, i) = \mathsf{val}(w, j)$ for all non-faulty i and k (that do not appear in w). The bad guys report at most f bad values $\mathsf{val}(wj, k')$, but the good guys report at least $n - f - |wj|$ good values $\mathsf{val}(wj, k)$. Since $n \geq 3f + 1$ and $|wj| \leq f$, we have $n - f - |wj| \geq 3f + 1 - f - f \geq f + 1$ good values, which are a majority. □

We call a node w **common** if $\mathsf{val}'(w, i) = \mathsf{val}'(w, j)$ for all non-faulty i, j. Lemma 10.2.2 implies that wk is common if k is non-faulty. We can also show that any node whose children are all common is also common, whether or not the last process in its label is faulty.

Lemma 10.2.3. *Let wk be common for all k. Then w is common.*

Proof. Recall that, for $|w| < f + 1$, $\mathsf{val}'(w, i)$ is the majority value among all $\mathsf{val}'(wk, i)$. If all wk are common, then $\mathsf{val}'(wk, i) = \mathsf{val}'(wk, j)$ for all non-faulty i and j, so i and j compute the same majority values and get $\mathsf{val}'(w, i) = \mathsf{val}'(w, j)$. □

We can now prove the full result.

Theorem 10.2.4. *Exponential information gathering using $f + 1$ rounds in a synchronous Byzantine system with at most f faulty processes satisfies validity and agreement, provided $n \geq 3f + 1$.*

Proof. Validity: Immediate application of Lemmas 10.2.1 and 10.2.2 when $w = \langle\rangle$. We have $\mathsf{val}'(j, i) = \mathsf{val}(j, i) = \mathsf{val}(\langle\rangle, j)$ for all non-faulty j and i, which means that a majority of the $\mathsf{val}'(j, i)$ values equal the common input and thus so does $\mathsf{val}'(\langle\rangle, i)$.

Agreement: Observe that every path has a common node on it, since a path travels through $f+1$ nodes and one of them is good. If we then suppose that the root is not common: by Lemma 10.2.3, it must have a not-common child, that node must have a not-common child, etc. But this constructs a path from the root to a leaf with no not-common nodes, which we just proved can't happen. □

10.2.2 Phase king gets constant-size messages

The following algorithm, based on work of Berman, Garay, and Perry [BGP89], achieves Byzantine agreement in $2(f+1)$ rounds using constant-size messages, provided $n \geq 4f+1$. The description here is drawn from [AW04, §5.2.5]. The original Berman-Garay-Perry paper gives somewhat better bounds, but the algorithm and its analysis are more complicated.

10.2.2.1 The algorithm

The basic idea of the algorithm is that we avoid the recursive majority voting of EIG by running a vote in each of $f+1$ *phases* through a **phase king**, some process chosen in advance to run the phase. Since the number of phases exceeds the number of faults, we eventually get a non-faulty phase king. The algorithm is structured so that one non-faulty phase king is enough to generate agreement and subsequent faulty phase kings can't undo the agreement.

Pseudocode appears in Algorithm 10.2. Each processes i maintains an array $\mathsf{pref}_i[j]$, where j ranges over all process ids. There are also utility values majority, kingMajority and multiplicity for each process that are used to keep track of what it hears from the other processes. Initially, $\mathsf{pref}_i[i]$ is just i's input and $\mathsf{pref}_i[j] = 0$ for $j \neq i$.

The idea of the algorithm is that in each phase, everybody announces their current preference (initially the inputs). If the majority of these preferences is large enough (e.g., all inputs are the same), everybody adopts the majority preference. Otherwise everybody adopts the preference of the phase king. The majority rule means that once the processes agree, they continue to agree despite bad phase kings. The phase king rule allows a good phase king to end disagreement. By choosing a different king in each phase, after $f+1$ phases, some king must be good. This intuitive description is justified below.

10.2.2.2 Proof of correctness

Termination is immediate from the algorithm.

For validity, suppose all inputs are v. We'll show that all non-faulty i have $\mathsf{pref}_i[i] = v$ after every phase. In the first round of each phase, process i receives at least $n-f$ messages containing v; since $n \geq 4f+1$, we have $n-f \geq 3f+1$ and $n/2+f \leq (4f+1)/2+f = 3f+1/2$, and thus these $n-f$ messages exceed the $n/2+f$ threshold for adopting them as the new preference. So all non-faulty processes ignore the phase king and stick with v, eventually deciding v after round $2(f+1)$.

```
pref_i[i] = input
for j ≠ i do pref_i[j] = 0
for k ← 1 to f + 1 do
    // First round of phase k
    send pref_i[i] to all processes (including myself)
    pref_i[j] ← v_j, where v_j is the value received from process j
    majority ← majority value in pref_i
    multiplicity ← number of times majority appears in pref_i
    // Second round of phase k
    if i = k then
        // I am the phase king
        send majority to all processes
    receive kingMajority from phase king
    if multiplicity > n/2 + f then
        pref_i[i] = majority
    else
        pref_i[i] = kingMajority
return pref_i[i]
```

Algorithm 10.2: Byzantine agreement: phase king

For agreement, we'll ignore all phases up to the first phase with a non-faulty phase king. Let k be the first such phase, and assume that the pref values are set arbitrarily at the start of this phase. We want to argue that at the end of the phase, all non-faulty processes have the same preference. There are two ways that a process can set its new preference in the second round of the phase:

1. The process i observes a majority of more than $n/2+f$ identical values v and ignores the phase king. Of these values, more than $n/2$ of them were sent by non-faulty processes. So the phase king also receives these values (even if the faulty processes change their stories) and chooses v as its majority value. Similarly, if any other process j observes a majority of $n/2+f$ identical values, the two $> n/2$ non-faulty parts of the majorities overlap, and so j also chooses v.

2. The process i takes its value from the phase king. We've already shown that i then agrees with any j that sees a big majority; but since the phase king is non-faulty, process i will agree with any process j that also takes its new preference from the phase king.

This shows that after any phase with a non-faulty king, all processes agree. The proof that the non-faulty processes continue to agree is the same as for validity.

10.2.2.3 Performance of phase king

It's not hard to see that this algorithm sends exactly $(f+1)(n^2+n)$ messages of 1 bit each (assuming 1-bit inputs). The cost is doubling the minimum number of rounds and reducing the tolerance for Byzantine processes. As mentioned earlier, a variant of phase-king with 3-round phases gets optimal fault-tolerance with $3(f+1)$ rounds (but 2-bit messages). Still better is a rather complicated descendant of the EIG algorithm due to Garay and Moses [GM98], which gets $f+1$ rounds with $n \geq 3f+1$ while still having polynomial message traffic.