Distributed Agreement

1. Introduction

These notes expand upon material in Chapter 15 of CDK (2nd. edition of *Distributed Systems – Concepts and Design*, by Coulouris, Dollimore and Kindberg), which examines fault tolerance in distributed systems. Chapter 15 presents Cristian’s classification of failures and the concept of masking failures using hierarchical and group techniques. The chapter also gives an informal account of the Byzantine Generals problem.

These notes consider further the problem of reaching agreement in the presence of faults. Examples of this general class of problem include transactional commit protocols – in which a collection of servers reach agreement on committing a set of updates, and clock synchronisation protocols – in which a collection of processes agree (albeit approximately) on the time of day. Protocols exist that are tailored to these individual types of agreement, but it is useful for us to consider agreement more generally, in a search for common properties and solutions.

We define three related agreement problems: Distributed Consensus (or Consensus) Byzantine Generals and Interactive Consistency. We define the system assumptions upon which they are based. Algorithms exist that solve the Byzantine Generals problem for a bounded proportion of ‘treacherous’ generals (faulty processes), but the solutions assume a synchronous distributed system. We go on to consider the well-known impossibility result of Fischer, Lynch and Patterson [1985], which states that in an asynchronous system a collection of processes containing only one faulty process cannot deterministically reach consensus. Finally, we consider how it is that practical algorithms exist for systems that do not meet the synchronous assumptions, despite the impossibility result.

2. The problems of agreement

Our system model includes a collection of processes $p_i, i = 1, \ldots, n$ communicating by message passing. For each problem we shall specify an assumption that up to some number $m$ of the $n$ processes are faulty – that is, they exhibit some specified types of fault. Section 2.5 of the accompanying notes on *Distributed System Models* gives a process- and channel-oriented classification of faults (rather than Cristian’s similar but service-oriented classification described in CDK Chapter 15). We call a process that suffers from no faults correct.

Our problem is to ensure that the correct processes agree on one or more values which one or more of the processes has proposed. For example, we require that all the correct computers controlling a spaceship’s engines decide ‘switch on’, or all of them decide ‘switch off’.

We model the processes as in Figure 1. A process uses an agreement layer, which exports two procedures: the application uses $\text{propose}(u)$ to propose a value of $u$ to the other processes; the agreement layer derives a single value $v$ or a vector of values (one for each process) from all the processes’ proposed values and supplies it to the application with $\text{decide}(v)$.

To help us to formulate our problem we use a convenient fiction that all faults occur in the agreement layer. That is, the application in each process correctly calls $\text{propose}(v)$ for some value

---

1Our problems also arise in the shared memory model, but we shall not discuss this here.
v. But the agreement layer may fail to send the value to other agreement-layer entities; or it may send different, arbitrary values – whatever the value \( v \) that the local application actually proposed.

There are several forms of the agreement problem, which differ according to the failures that are allowed, the values that may be proposed, and how many processes propose values. Here we shall examine Byzantine Generals, Distributed Consensus and Interactive Consistency and the relationships between them. Hadzilacos and Toueg [1994] relate the problem of atomic and totally ordered multicast to the Distributed Consensus problem.

**Byzantine Generals (BG)**

Here a single process (the ‘commander’) proposes a single value, drawn from a set \( D \). The goal of the algorithm is for the correct processes amongst the others (the non-commander processes are all ‘lieutenants’) to agree on a common value, the *decision value*. The requirements are:

- **Termination.** All correct processes eventually choose a single decision value.
- **Agreement.** The decision value of all correct processes is the same.
- **Validity.** If the commander is correct, then all correct processes decide on the value that the commander proposed.

Note that Validity implies Agreement when the commander is correct; but the commander need not be correct.

**Distributed Consensus (DC)**

In contrast to Byzantine Generals, here *every* process proposes a single value, drawn from a set \( D \). The goal of the algorithm is for the correct processes to agree on a single value. The requirements are:

- **Termination.** All processes eventually choose a single decision value.
- **Agreement.** The decision value of all correct processes is the same.
- **Validity.** If all correct processes propose an identical value \( v \), then the decision value is \( v \).

Note that Validity allows a collection whose correct processes propose differing values in the set \( \{0, 1\} \) to decide upon the value 99! Neiger [1993] defines an alternative requirement called *Strong Validity*:

- **Strong Validity.** The decision value is the same as that proposed by some correct process.

Note that Strong Validity implies Validity, but the converse does not hold. Strong Validity is a more useful property than Validity, but we shall use the Validity requirement in our definition of DC.
Interactive Consistency (IC)\(^2\)

In the interactive consistency problem every process proposes a single value, drawn from a set \(D\). The goal of the algorithm is for the correct processes to agree on a vector of values, one for each process. We shall call this the ‘decision vector’. The requirements are:

- **Termination.** All processes eventually choose a vector.
- **Agreement.** The decision vector of all correct processes is the same.
- **Validity.** If \(p_i\) is correct and \(p_i\) proposes \(v_i\), then all correct processes decide on \(v_i\) as the \(i\)'th component of their vector.

Relationships between the problems

This section shows that we can sometimes derive a solution to one problem using a solution to another. This is a very useful property, both because it increases our understanding of the problems, and because by re-using solutions we can potentially save on implementation effort and complexity.

We characterise solutions to our three problems – without stating what those solutions are – as follows. The solutions satisfy the relevant versions of Termination, Agreement and Validity, but they are not necessarily deterministic. In principle they could produce different decision values for the same set of proposed values in different failure circumstances. We define the three solutions to hold for the same set of system assumptions. When, for example, we prove that we can solve IC given a solution to BG, we mean: we can do this given the same failure assumptions, and the same assumption that the system is synchronous or asynchronous.

\(BG_i(j,v)\) returns a decision value made by \(p_i\) according to the requirements of the Byzantine Generals problem, where \(p_j\), the commander, proposes the value \(v\).

\(DC_i(v_1...v_n)\) returns a decision value made by \(p_i\) according to the requirements of the Distributed Consensus problem, where \(p_1\) proposed \(v_1\), \(p_2\) proposed \(v_2\) etc.

\(IC_i(v_1...v_n)[j]\) returns the \(j\)'th value in the decision vector of \(p_i\) according to the requirements of the Interactive Consistency problem, where \(p_1\) proposed \(v_1\), \(p_2\) proposed \(v_2\) etc.

We construct solutions out of the solutions to other problems as follows.

**IC from BG.** We construct a solution to IC from BG by running BG \(n\) times, once with each process \(p_i, i = 1...n\) acting as the commander:

\[IC_i(v_1...v_n)[j] = BG_i(j,v_j).\]

The Termination, Agreement and Validity conditions for IC follow immediately from those for BG.

**DC from IC.** We construct a solution to DC from IC by running IC to produce a vector of values at each process, then applying an appropriate choice function on the vector’s values to manufacture a single value:

\[DC_i(v_1...v_n) = \text{majority}(IC_i(v_1...v_n)[1]...IC_i(v_1...v_n)[n])\]

The function \(\text{majority()}\) returns the majority value out of \(u_1...u_n\), if this exists, or a distinct value \(\bot\) if no majority value exists. In particular, \(\text{majority}(u_1...u_n) = u\) if \(u_1 = ... = u_n = u\) (that is, to satisfy

\(^2\)Lamport et al [1982] refer to ‘interactive consistency conditions’ in the context of the Byzantine Generals problem, but these are not to be confused with the Interactive Consistency problem.
Validity for DC). Termination, Agreement and Validity follow for DC from the same properties for IC and the deterministic and functional character of \textit{majority}.

\textbf{BG from DC.} We can also construct a solution to BG from DC as follows.

- The commander $p_j$ sends its proposed value $v$ to itself and each of the remaining processes
- All processes run DC with the values $v_1, \ldots, v_n$ that they receive
- We let $BG_i(j, v) = DC_i(v_1, \ldots, v_n)$.

Note that the values $v_1, \ldots, v_n$ may differ from $v$, because the commander’s agreement layer may be faulty and send different values to different processes. Termination and Agreement for BG follow automatically from Termination and Agreement for DC. To see that Validity for BG holds, note that if the commander is correct then $v_1 = \ldots = v_n = v$; by Validity for DC, $BG_i(j, v) = v$ as required.

In summary, given a solution to BG we may solve both IC and DC ($BG \rightarrow IC \rightarrow DC$); given DC, we may solve BG.

\textbf{3. The Byzantine generals problem}

We study the BG problem for $n$ processes. Our failure assumption is: up to $m$ processes may be faulty, and they exhibit Byzantine failure. That is, they may send any message with any value at any time; and they may omit to send any message. We assume further that:

- There is a direct, private channel between each pair of processes
- Channels deliver all messages correctly as sent
- A process knows the identity of every received message’s sender
- A process can detect the absence of a message (through a time-out)

The privacy of channels is important to our problem statement. If a process could see all the messages that another process sends, then it could detect another process’s inconsistencies and easily deduce that it is faulty. Another important factor is whether the messages are \textit{signed}. A receiver can tell whether a value is signed by a particular process, and it can tell the value that it signed. Signatures cannot be forged.

We shall give the presentation of Lamport, Shostak and Pease [1982] for unsigned (they call them ‘oral’) messages. It was they who first formulated the problem in terms of the analogy of Byzantine generals. The commander among them is supposed to issue a command to his lieutenants – for example, to retreat or attack. The lieutenants are supposed to act upon the order together. It is known, however, that one or more of the generals may be treacherous, and if so will try to fool the loyal generals into acting differently by sending inconsistent values; the loyal generals are to use private channels to agree on a uniform action despite this threat.

Lamport, Shostak and Pease began by showing that there is no solution to BG with unsigned messages for three processes, with one of them faulty. They generalised this result to show that no solution exists if $n \leq 3m$. They went on to give an algorithm that solves BG if less than a third of the processes are faulty, that is, if $n \geq 3m+1$.

\textbf{Impossibility with three processes}

We now show that it is impossible to solve the BG problem with three processes, one of which is faulty.

Figure 2 on the left shows a configuration in which one of the ‘lieutenants’, $p_3$, is faulty; on the right the commander, $p_1$, is faulty. Each drawing in Figure 2 shows two rounds of messages: the values the commander sends, and the values that the lieutenants subsequently send to one another. The numeric prefixes serve to specify the sources of messages and to show the different rounds. Read the ‘:’ symbol in messages as ‘says’; for example, ‘3:1:u’ is the message ‘3 says 1 says u’.

© Tim Kindberg
In the left-hand scenario the commander correctly sends the same value $v$ to each of the other two processes, and $p_2$ correctly echoes this to $p_3$. However, $p_3$ sends a different value $u$ to $p_2$. All $p_2$ knows at this stage is that it has received differing values; it cannot tell which it should choose.

In the right-hand scenario the commander is faulty and sends differing values to the lieutenants. After $p_3$ has correctly echoed the value $u$ that it received, $p_2$ is in the same situation as it was in when $p_3$ was faulty: it has received two differing values.

If a solution exists, process $p_2$ is bound to decide on value $v$ when the commander is correct – by the Validity condition – and since it cannot tell the difference it must choose $v$ in the right-hand scenario also.

If we follow exactly the same logic for $p_3$, we see that $p_3$ also has to decide on the value that the commander sends to it. But this contradicts the Agreement condition (the commander sends differing values if it is faulty). So no solution is possible.

Note that this ‘proof’ rests on our intuition that nothing can be done to improve a correct lieutenant’s knowledge beyond the first stage, where it cannot tell which process is faulty. It is possible to prove the correctness of this intuition [Pease et al 1980]. In the appendix we give a proof of the general result that a solution to BG is impossible with $n \leq 3m$ processes.

**Solution with $n \geq 3m+1$ processes**

The algorithm of Lamport, Shostak and Pease proceeds in rounds of messages. Before describing the general algorithm, we analyse the case of $n \geq 4$ processes, with just one process faulty. We assume for the moment that a faulty process always sends a value – it never omits to send a message.

It is possible for the correct lieutenants to reach agreement in just two rounds of messages:

- In the first round, the commander sends a value to each lieutenant
- In the second round, each lieutenant sends the value it received to each of the other lieutenants.

A lieutenant receives a value from the commander, plus $n-2$ values from its fellow lieutenants. If the commander is faulty, then all the lieutenants are correct and each will have gathered exactly the set of values that the commander sent out. Otherwise one of the lieutenants is faulty. Each correct lieutenant gathers $n-2$ copies of the value that the commander sent, plus a value that the faulty lieutenant sent to it.

We now illustrate the algorithm that we have just described for the case of four generals. It uses the same function `majority()` that we introduced above when we derived a solution to DC from a solution to IC.
In the left-hand case of Figure 4 lieutenant $p_3$ is faulty:

$p_2$ decides on $\text{majority}(v,u,v) = v$

$p_4$ decides on $\text{majority}(v,v,w) = v$ – they agree, deciding on the commander’s value.

In the right-hand case of Figure 4 the commander is faulty:

$p_2$, $p_3$ and $p_4$ decide on $\text{majority}(u,v,w) = \bot$ – they agree.

We have proved that this is a correct algorithm for no more than one process faulty, $n \geq 4$ and with no message omissions. It is a particular case of the general solution to BG of Lamport, Shostak and Pease, which requires one more round of message passing than the number of faulty processes – that is, it needs $m+1$ rounds. So if there can be two faulty processes we need three rounds; three faulty processes require four rounds, etc.

In the general algorithm we account for the fact that a faulty process can omit to send a message. If a correct process does not receive a message within a suitable time limit, it proceeds as though the faulty process had sent it the value $\bot$.

We define the protocol recursively, so that the protocol unfolds in $m+1$ stages, numbered 0, 1, 2, ..., $m$. We can picture the unfolding as a tree (Figure 5). At stage $s$ of the protocol, the agreement layer of each of a set $L$ of $n-1-s$ lieutenants receives a value from a process $c$ (the commander in this stage). At the next stage, each member of $L$ adopts the role of commander and sends a value to the other processes in $L$.
Let $\text{valueFrom}(s, c, p)$ be the value the agreement layer of process $p \in L$ receives from process $c$ at stage $s$, or $\perp$ if $p$ times out. The complete protocol is made up of three declarations, as follows. We assume that $m$, the maximum number of faulty processes, is globally defined in these declarations:

1. The final value delivered to application at a given process $\hat{p} \in \hat{L}$ (a set of lieutenants) from commander $\hat{c}$ is $\text{BGDeliver}(0, \hat{c}, \hat{p}, \hat{L})$.

2. $\text{BGDeliver}(s, c, p, L)$ ($s < m$) = $\text{majority}(\text{valueFrom}(s, c, p),$
   \[ \text{BGDeliver}(s+1, q, p, L-\{q\}), \]
   \[ \text{BGDeliver}(s+1, q', p, L-\{q'\}), \]
   \[ \ldots \]

   where $\{q, q', \ldots\} = L-\{p\}$.

3. $\text{BGDeliver}(m, c, p, L) = \text{valueFrom}(m, c, p)$.

In other words, the protocol $\text{BGDeliver}(m, c, p, L)$ gives process $p \in L$ whatever value it receives from $c$. The protocol $\text{BGDeliver}(s, c, p, L)$ ($s < m$) gives process $p \in L$ the majority value from the value received directly from $c$, together with the $|L|-1$ values delivered by running the next stage of the protocol for each other lieutenant in $L$ acting as commander. The agreement layer holds back the results of each stage until it can produce $\text{BGDeliver}(m, \hat{c}, \hat{P}, \hat{L})$, the final value, which it passes on to the application.

To aid in understanding, the reader is advised to work out with pencil and paper the messages sent in this protocol for $m = 2$. The full proof of the algorithm’s correctness is beyond the scope of these notes, but can be found in [Lamport et al 1982].

**Protocol complexity**

We can measure the optimality of a solution to BG by asking:

- How many message rounds does it take? (This is a factor in how long it executes for.)
- How many messages are sent, and of what size? (This measures the total bandwidth utilisation, and has an impact on the execution time.)
- How many processes are required if up to $m$ are faulty? (How many redundant processors must we buy?).

The Lamport, Shostak and Pease protocol requires $m+1$ rounds. As Figure 5 above shows, the 0th stage accounts for $(n-1)$ messages; the 1st stage accounts for $(n-1)(n-2)$ messages, the 2nd stage accounts for $(n-1)(n-2)(n-3)$ messages. This pattern repeats until the $m$th stage. The protocol is therefore $O(n^{m+1})$ in the number of messages it sends.

Fischer and Lynch [1982] proved that any deterministic solution to BG will take at least $m+1$ message rounds – so no algorithm can operate faster in this respect than that of Lamport, Shostak and Pease. However, there are algorithms that require fewer messages. For example, Dolev and Reischuk [1985] give an algorithm that is $O(nm + m^3)$. Unfortunately, this algorithm has worse time complexity: it requires $2m+3$ message rounds. Garay and Moses [1993] designed an algorithm which requires the minimum value of $m+1$ rounds, but which uses a number of messages that is polynomial in the number of processes.
Signed messages

In the BG problem as we have studied it so far, a faulty process is able to make the (implicit) statement that another process sent it any value it chooses. This is not so if we use digitally signed messages (see CDK Chapter 16). Using signed messages reduces the difficulty of the BG problem. No process may forge another process’s signature. If a faulty process does not sign its values correctly according to the chosen digital signature scheme, then receivers may observe this and deduce that the process is faulty. And if a faulty process does correctly sign its values, but signs and sends different values to different processes, then by exchanging messages the destination processes can readily spot the use of different values.

Several algorithms, such as that of Dolev and Strong [1983], take advantage of signed messages. Dolev and Strong’s algorithm again has a time complexity of $m + 1$. But the number of messages sent is only $O(n^2)$, instead of $O(n^{m+1})$.

To aid understanding, the reader should show that a solution to BG with signed messages does exist for $n = 3, m = 1$ – unlike the case of unsigned messages.

4. Distributed Consensus

We showed in Section 2 that we can construct a solution to DC, if we are given a solution to BG. We have given a solution to BG, and so there is a solution to DC. However, in discussing solutions to BG we concerned ourselves only with synchronous distributed systems. The algorithms assume that message exchanges take place in rounds, and that processes are entitled to time out and assume that a faulty process has not sent them a message, because the maximum possible delay has been exceeded.

Fischer, Lynch and Paterson [1985] demonstrated that Distributed Consensus has no solution in an asynchronous system, even if only at most one process is faulty, and even if the process fails simply be stopping sending any more messages. This result does not mean that processes can never reach distributed consensus in an asynchronous system if one is faulty. It means that no deterministic algorithm exists that can guarantee consensus will be reached. Their proof involved showing that there is always some run of the processes’ execution that avoids consensus being reached.

We immediately know from the result of Fischer et al that there is no solution to BG in an asynchronous system, because if there were we would have a solution to DC.

Despite the result of Fischer et al, practical algorithms exist which find agreement even in systems with unbounded message delays. This is possible because in these algorithms the processes agree to deem a process that has not responded for more than a bounded time to have failed. That is, they make the implicit assumption that the system is synchronous. An unresponsive process may not really have failed but they act as if it had done. They make the failure fail-silent by discarding any subsequent messages that they do in fact receive from a ‘failed’ process. For example, this technique is used in the ISIS system [Birman 1993].

5. Summary

These notes have described the three problems of Distributed Consensus, Interactive Consistency and Byzantine Generals. We defined the conditions for their solution and we showed relationships between these problems.

Several solutions exist for BG in a synchronous system. We described the solution due to Lamport, Shostak and Pease. More recent algorithms have lower complexity, but in principle none can better the $m+1$ rounds taken by this algorithm. Every solution to BG requires the number of faulty processes to be less than a third of the total.

We described a result that a deterministic solutions to Distributed Consensus is fundamentally impossible in an asynchronous distributed system.

Tim Kindberg, QMW
References (not in CDK)


![Figure 3. Simulating n n-generals with three 3-generals.](image_url)

**Appendix: Impossibility of solution to BG with n ≤ 3m processes**

We use the impossibility result with three processes, one of them faulty, to show that in general no more than a third of the processes can be faulty if BG is to have a solution.

To do this, we suppose that there is a solution with n ≤ 3m. We shall show that this assumption implies that there is a solution to the problem with n = 3, m = 1, which we know is impossible. We shall thus conclude that the assumption is false – there is no solution for n ≤ 3m.

© Tim Kindberg
Suppose we are given three Byzantine generals, which we call 3-generals. At most one of them is incorrect. We assign to these three processes that task of simulating n generals running the hypothesised solution protocol for BG (see Figure 3). We call these the n-generals. By 'simulate' we mean that a 3-general sends messages on behalf of its n-generals. In particular, the 3-commander simulates the n-commander. A 3-general sends some messages internally to the n-generals that it simulates, others it sends over external channels to n-generals simulated by other 3-generals. A faulty 3-general simulates faulty n-generals and sends spurious messages on behalf of them; a correct 3-general simulates correct n-generals.

We assign the commander the task of simulating \( n_1 \) n-generals (including the n-general commander) and to the lieutenants the task of simulating \( n_2 \) and \( n_3 \) n-generals, with \( n_1 + n_2 + n_3 = n \) and \( n_1, n_2, n_3 \leq m \) – that is, \( n \leq 3m \), and at most \( m \) of them are faulty (those simulated by the faulty 3-general). We now apply our hypothesised solution for n generals. We map this onto a solution for the 3-generals as follows:

- the value first sent by the 3-commander to a 3-lieutenant is the first value it sends to it as part of its simulation of the n-commander
- a correct 3-lieutenant decides on the common value that its (correct) n-generals decide.

It is straightforward to show that Termination, Validity and Agreement hold for the 3-generals given the supposed BG-solution for the n-generals, and we leave this to the reader. We have established a solution to BG that we know to be impossible, and the hypothesised solution is false.