Design and Analysis of Algorithms CSCI 5454
Crowd Source Lectures 28 March 2011
Charles Chen and Rowan Wing

1 Byzantine Agreement

The name of this topic comes from the "Byzantine Generals' Problem", which was proposed in 1980 by Marshall Pease, Robert Shostak, and Leslie Lamport [1]. The classic problem details the difficulty that the Byzantine army experiences during an attack on an enemy city. Various portions of the army have made their way to disconnected positions around the city, and must coordinate their actions in order to successfully invade. Some of the key details:

1. The Byzantine army consists of multiple divisions.
2. Each division of the army is directed by its own general.
3. Generals communicate with each other via messengers (pairwise communication).
4. Generals are either loyal or traitorous.
5. Loyal generals need to agree on the course of action to succeed.
6. A small number of traitors will try prevent the loyal generals from reaching an agreement.

Given this set of conditions, we have the following problem: what set of protocols can we devise that allows for loyal generals to reach a consensus, when we don't know the loyalties of all the other participants?

1.1 Motivation

Solutions to this problem have many practical applications in the context of distributed systems, where coordination between many units is necessary. A loss of coordination due to random breakdowns, connection failures, or even malicious interference can cause entire systems falling apart if adequate procedures are not implemented. This is exactly what happened to Amazon's online storage service in 2008, when it was temporarily disabled by a single-bit hardware error that propagated through the system[2]. Byzantine fault tolerant protocols are particularly important in web services, where you may have various servers that need to be checked for faults during operation.

In the context of a distributed system, Byzantine faults consist of both:

1. Omission failures: crash failures, failures to receive a request, failures to send a response

2. Commission failures: incorrect processing, corruption of local states, sending incorrect responses

Faults may arise due to variety of complications, such as hardware failures, network congestion, and malicious attacks. A Byzantine fault tolerant algorithm can cope with these situations by minimizing the effects of faulty units and allowing the rest to operate properly in spite of them.

Byzantine Agreement algorithms are also useful for situations needing agreement, such as in a leader election problem. Such an algorithm would allow loyal participants to come to a consensus on who their leader is, despite traitorous elements trying to convince them otherwise.

1.2 The Simplified Problem

Pease and his colleagues noted that this problem can be reduced to a series of problems involving just a single commanding general and several lieutenants[4]. The original requirement that all loyal generals agree on a course of action can be seen as a need for all generals to act upon the same information in the same manner. As generals obtain information from each other, it is necessary for a single general to give information to all $n-1$ others. The other generals must then act upon that information in the same way (provided they are loyal).

When this occurs, it is as though that general is issuing a command to the others (information leading to predetermined action vs. a commanding an action). Designating the subordinates as lieutenants, the problem is transformed into a matter of communication between a potentially traitorous commanding general and potentially traitorous lieutenants.

Solutions to this variation of the problem must satisfy the following:
A1: If the general is loyal, then all loyal lieutenants will follow his given order.
A2: Loyal lieutenants will all obey the same order

The first condition enforces that loyal lieutenants will behave as the loyal general commands. The second requires that the commanding general be loyal or, in the absence of that, that all loyal lieutenants default to the same course of action.

From the point of view of any participant, it is difficult to determine which of the others are loyal. As communication is done via messengers, it is possible for messages to and from lieutenants to be lost in transit. It is also possible for traitors to attempt to mislead loyal generals by sending forged messages, trying to appear loyal or to paint other participants as traitorous. The issue is then to create a secure and structured communication protocol that can convey the loyalty and the orders of individual units to each other.

1.3 Solution with Oral Communication

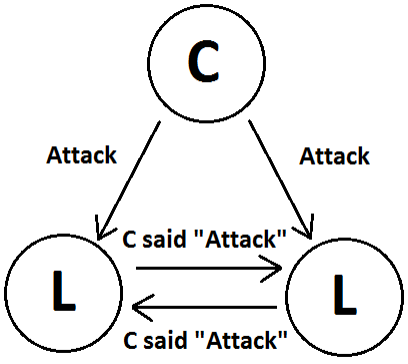
Before we consider the problem more thoroughly, it is necessary to define the type of communication that allows for this particular problem. In this situation we only consider oral communication, defined as communication where the contents of messages are solely determined by the sender. This allows the sender to transmit any message that it desires, allowing traitors to 'lie'.

This communication model has some key assumptions:
B1: Messages are delivered correctly
B2: The sender is known to the receiver
B3: The absence of a message can be detected [4]

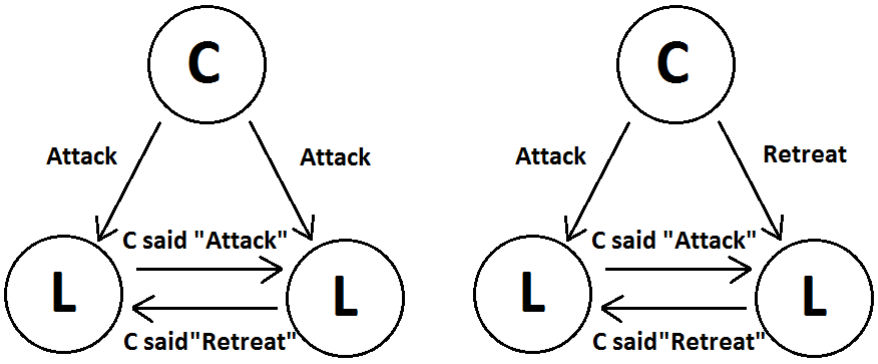
The first allows for an optimistic situation where messages are delivered as intended by the sender. B2 prevents senders from pretending to be other participants. B3 prevents a situation where a participant waits for a response from another participant indefinately.

Within this oral communication framework, it is necessary for lieutenants to relay the orders that they received to the other lieutenants. This verifies that they are all receiving the same order, and thus allows for them to carry out the same action.

Consider a simple case involving a single commander and two lieutenants, with all participants loyal. The commander would send a request to both lieutenants. To ensure that they follow the same command as each other, they compare the requests that they received from the commander. Since the commander is loyal, they are the same. The two lieutenants can proceed to execute the commander's request.



Now consider a situation where there is at least one traitor in the group of three. Let's say that a loyal lieutenant receives a request from the commander stating that he should attack. Let's also say that the other lieutenant sends the loyal lieutenant a message stating that the commander told him to retreat. In this situation, the loyal lieutenant cannot determine which of the others are traitors, as both situations are possible.



In this situation, the loyal general does not know what course of action should be taken, illustrating that this variant of the Byzantine General's problem has a lower limit of solvability. In the 3-participant situation with a single traitorous lieutenant, the loyal lieutenant does not know if the commander is loyal or not, and thus cannot follow his order. This violates the previously described requirement A1, that loyal lieutenants follow the loyal commander's orders. For a byzantine agreement problem where only oral messages are allowed, a solution only exists if there are a maximum of t traitors for a group of participants of $3t + 1$.

Based on these two examples, we can now consider an algorithm that achieves Byzantine agreement. From the previous example, we see that one of the biggest problems with oral communication is the ability for participants to 'lie'. This makes it necessary to verify what each participant to check what everyone else is telling them. A good way to do this is by doing a sort of "verification by gossip". Instead of having just two rounds of communication, with the first verifying the second, we can have multiple rounds, each verifying the information in the last round. This would be like a situation in which a participant goes, "Lieutenant 3 told me that the commander said to attack. I'm going to ask the other lieutenants what they heard from Lieutenant 3."

For a problem with n participants and at most $t = \lfloor \frac{n-1}{3} \rfloor$ traitors, where $n \geq 3t + 1$, we have the following recursive algorithm:

Oral_Byzantine(t), $t = 0$:
1) The commander sends his order to every lieutenant
2) The lieutenants accepts the order of the commander, or accepts "Retreat" as a default in the absence of a commander's order

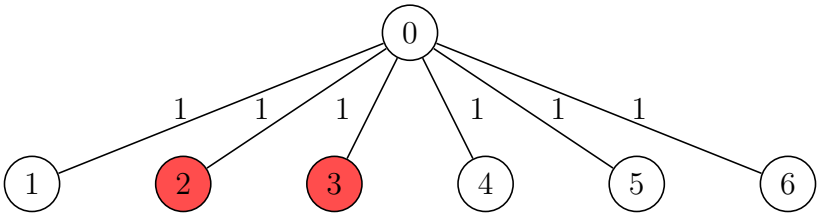
Oral_Byzantine(t), $t > 0$:
1) The commander sends his request to all lieutenants
2) For each lieutenant i, let v_i be the message that i receives from the commander in the previous action. Each i acts as a commander and sends its v_i to each of the $n - 2$ other lieutenants using algorithm Oral_Byzantine($t - 1$).
3) For each lieutenant i and each other lieutenant $j \neq i$, let v_j be the message that i

receives from j. Lieutenant i uses the value of $Majority(v_1, v_2, ...v_{n-1})$ as the action to be taken.

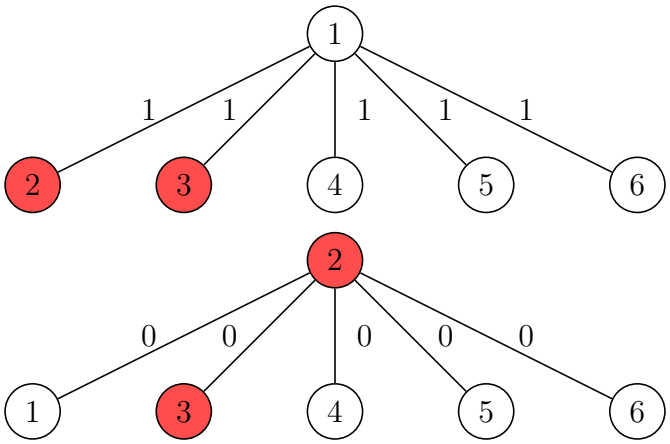
Here, if the majority of values $v_1, v_2...v_{n-1}$ are equal to v, then $Majority(v_1, v_2, ...v_{n-1})$ returns v, or "retreat" if no majority exists.

Observe the following example with $n = 7$ participants, $t = 2$ of whom are traitors. We'll label the commander '0' and the lieutenants 1 through 6. For simplicity we'll mark "attack" as "1" and retreat as "0". In this situation, we'll only have lieutenants 2 and 3 be traitors.

First, the commander sends his request to all the lieutenants as part of Oral_Byzantine(2).



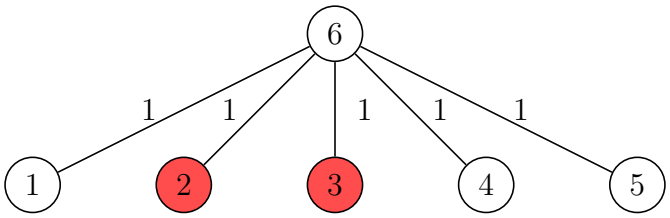
Now, each lieutenant acts as a commander towards all the other lieutenants and performs Oral_Byzantine(1). Since 2 and 3 are traitors, they will try to get the others to retreat.



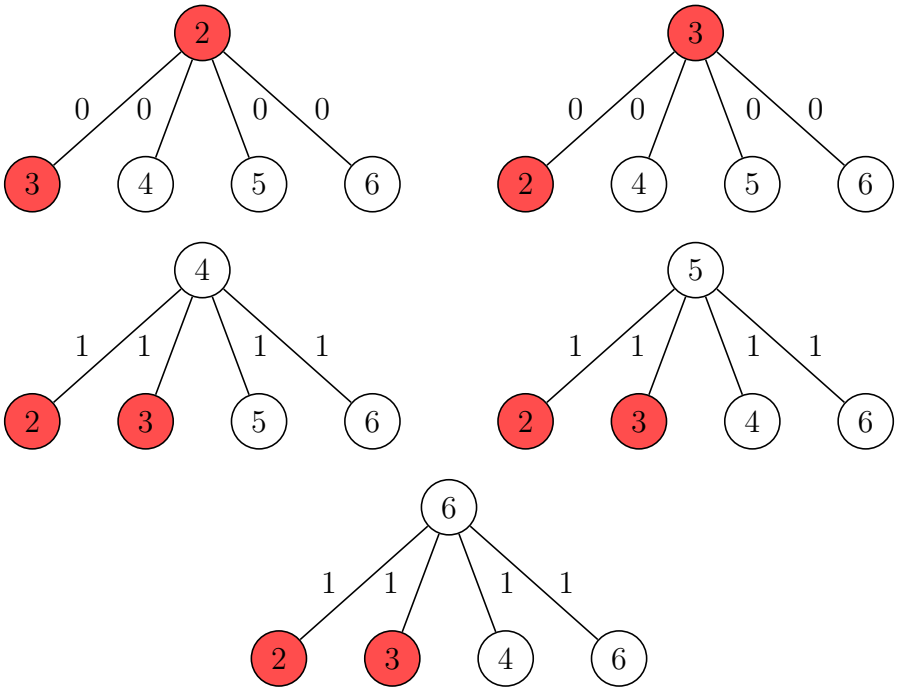
...

...

...

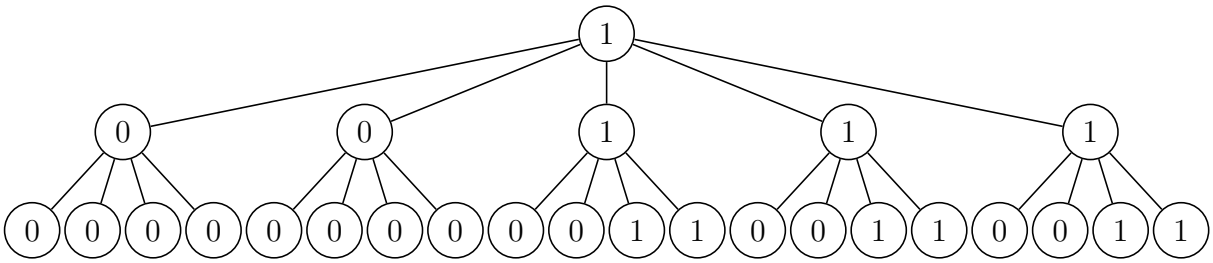


For each 'sub-commander', its lieutenants each run an instance of Oral_Byzantine(0) with each of the other lieutenants under the 'sub-commander'. Here we observe just the result for sub-commander 1's lieutenants.



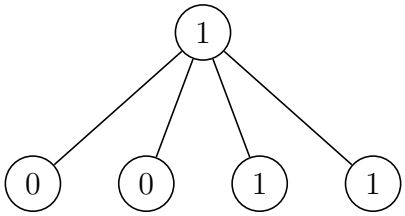
The previous trees illustrated the sequence of message transmissions. To better understand the Majority-taking of step 3 in the Oral_Byzantine(t) algorithm, we can use the receive-tree resolution method described by Bar-Noy et al [5]. Note that the notation here is different from that of the 'received message' tree.

When the commander sends his request to a lieutenant during Oral_Byzantine(t), this value is placed at the root of a tree. The values sent to this lieutenant in subsequent calls of Oral_Byzantine($t-1$) are placed as children under that node, and calls of Oral_Byzantine($t-2$) are placed as children under those nodes, etc. In this tree, the root node shows what the particular lieutenant received from the commander. The children of that node indicate what the other lieutenants say they receive from the parent node. The below is the 'received' tree for lieutenant 1:

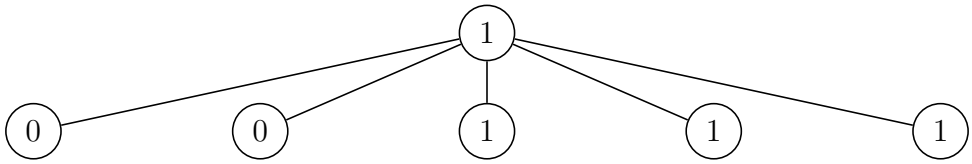


Each tier in the tree represents the set of values that the owner of the tree received from all other participants during a particular phase of messaging. The root node corresponds to the value received during Oral_Byzantine(t), the second tier represents the values received during Oral_Byzantine($t-1$)... etc.

To perform step 3 of the algorithm, we take each bottom most 2-tier subtree and find what the majority of the values are, and replace the root of that subtree with the majority. In the absence of a majority, we place a default value (here 0) in its place. The rightmost subtree is shown below:



Note that the majority of the vertices in this subtree have the value "1", so we can collapse the entire thing into just a single vertex of value "1". Doing the same for the other bottom-most subtrees, we have the following:



We can collapse this smaller tree into a single majority value. This value is what lieutenant 1 considers to be the order from the commander. Since lieutenants 4, 5, and 6 are also loyal, their trees should show a similar amount of misinformation from traitors 2 and 3, and come to the same conclusion. The algorithm ends with lieutenants 1, 4, 5, and 6 executing order 1 (attack), with 2 and 3 doing whatever they want because they're traitors.

The received message tree demonstrates the correctness of the algorithm. Each bottom most subtree, prior to being collapsed, represents a smaller instance of the original problem. With each call of Oral_Byzantine($t-1$), the number of participants decreases by one. In the worst case, we always end up excluding a loyal participant.

Since we start with a minimum of $2t + 1$ loyal components outnumbering t traitors, in the last call of the algorithm we would still have $t + 1 > t$, with the loyal components outnumbering the traitors. Because of this we can simply take the majority and collapse the tree upward, recursively deriving a consensus. This is comparable to the solution of the "Faulty Processor" problem we had in problem set 1.

By counting the number of edges in the sent-message tree, we see that the total number of messages sent is $O((n-1)(n-2)...(n-t)) = O(n^t) = O(n^{\frac{n}{3}})$. The protocol takes exponential messaging, which is less than ideal.

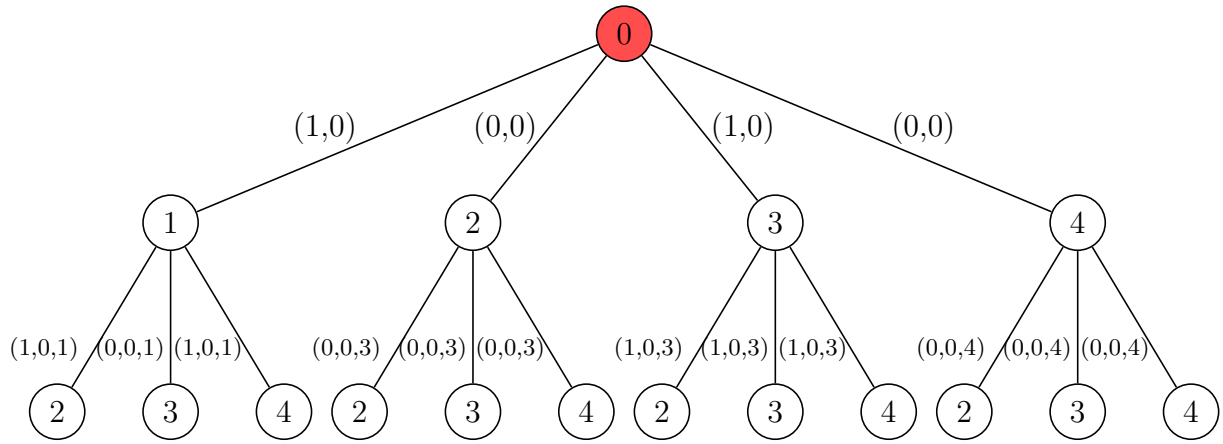
1.4 Solution with Signed Messages

A method that restrains the traitors' ability to lie was proposed by three researchers of SRI International [4], which entailed using a signed message protocol. This algorithm first has the commanding general send his signed order to all lieutenants. The lieutenants then recursively append the received order with their own signature and forward that message to all other lieutenants. If a loyal general's signature cannot be forged, then traitorous lieutenants must either forward the loyal general's message as is, or attempt to forge the signature and be discovered. This, however, does not prevent a traitorous general from sharing his signature with traitorous lieutenants in order to collude against the others.

Consider a simple situation with only two orders, "attack" and "retreat". When a lieutenant receives an message for the first time, he records the order received in an internal set, then broadcasts a signed duplicate of the message. If the lieutenant receives further messages of the same order, it ignores them. If the lieutenant receives a properly signed message with a different order, then this message is recorded in the set and a signed duplicate of the message is again broadcasted to the rest of the lieutenants.

At the end of the messaging phase, each lieutenant has one of three possible values for their "received order" sets: {attack}, {retreat}, and {attack, retreat}. The first two cases will only occur if the commander is loyal. The last case will occur only if the commander distributes properly signed but contradictory orders to the lieutenants, indicating that the commander is a traitor. In this situation the lieutenants take some default action.

An example is shown below, with a traitorous commander. Here we use the following edge notation: (*order, signature 1, signature 2*).



With the guarantee of message validity that signing gives, we don't need as many messaging steps as the previous unsigned algorithm. As a result, only $1 + (n-1)(n-2) = O(n^2)$ messages are exchanged. This is a substantial improvement over the exponential time that we had previously.

This protocol requires two things to be true:

1. Loyal participants have signatures that cannot be forged
2. Any participant can verify the authenticity of a signature

In practice, these two can be accomplished using public key cryptography. Each participant would sign sent messages with his private key, which every other participant can verify using the corresponding public key.

2 Practical Byzantine Fault Tolerance

Since the initial posing of the Byzantine Generals' Problem in 1980 there has been a multitude of protocols built around the general principles of Byzantine Agreement. One such protocol was developed in 1999 by Miguel Castro and Barbara Liskov and is described in their paper "Practical Byzantine Fault Tolerance" [8]. The Practical Byzantine Protocol was designed to work on real systems and to be both efficient and robust in the presence of Byzantine faults. It was a landmark development that not only showed significant (an order of magnitude) improvement in speed over earlier algorithms, viz. Ramport and SecureRing, but it was also the first protocol to work in an asynchronous environment [8]. Developing a protocol that worked in asynchrony was noteworthy because the assumption of synchrony for reliability can be dangerous in the presences of malicious attacks. Malicious attacks can exploit synchrony by delaying non-faulty nodes or the communication between them until they are deemed faulty and discarded. Asynchrony is also necessary for many real-time distributed systems like file systems and the internet where many clients are simultaneously accessing the same information.

2.1 Assumptions

The Practical Byzantine Protocol makes a number of assumptions to ensure the algorithm functions properly.

1. All elements of the distributed system are state-machine replicas. This means that each element duplicates the same initial state and runs the same version of the service code.
2. The replicas are deterministic, that is they produce the same result given the same initial state and arguments
3. Independent replica failures are also assumed, which requires that each replica has its own operating system, password, and administrator.
4. Messages are signed by the sender and contain public-key signatures, authentication codes, and message digests[1]. All replicas are also able to verify the signatures of the other replicas.
5. The time between when a message is sent and when it is received is not indefinite if messages are retransmitted until they are received.
6. A strong adversary is assumed that is able to coordinate the faulty nodes, but which is still computationally bound and unable to break the public-key encryption of the message signatures or decipher the message digests.
7. Since any protocol can be overrun by the presence of an extreme number of faulty nodes, the best one can expect is a protocol that is resilient in the presence of some number f of expected faults [6]. In this protocol the maximum number of expected faulty nodes is $\lfloor \frac{n-1}{3} \rfloor$.

2.2 Limitations

The algorithm does not require synchrony for reliability, but does require it for liveness, i.e. clients eventually receive replies to their requests. This is a fairly weak assumption of synchrony that is likely to be found in real systems where faults are eventually repaired. This assumption however guards against the impossibility of a fully asynchronous environment as described in "*Impossibility of Distributed Consensus with One Faulty Process*" [6]. The algorithm also does not address fault-tolerant privacy and faulty nodes may leak information to the adversary (Secret sharing schemes may be found in future work that allows the algorithm to maintain fault-tolerant privacy even at the threshold of faulty nodes.)

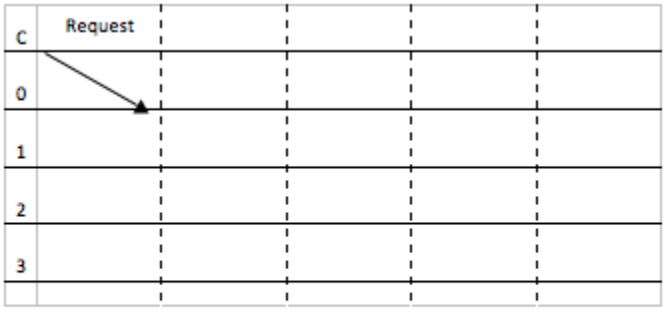
2.3 Overview of Algorithm

[1] A message digest is a cryptographic hash value that represents a block of data. Message digests are used to authenticate data since any change to the data will also change the message digest. This is very similar to parity and checksums.

The Practical Byzantine Protocol is a signed message protocol, as described above, that ensures that each replica cannot send a faulty message without being discovered. It works in five rounds. During each round there are interactions between the client, the primary replica, and the backup replicas. The client is the end user who is making a request of the system. This could be a actual person who is utilizing a web service or retrieving a file from a file system or some other computerized agent interacting with the distributed system. The primary replica is the replica that has been designated to take requests from the client and takes on a role similar to the general's role in the simplified problem above. There is nothing unique about the primary replica in terms of state or function and any replica may be designated as a primary. The backup replicas are all the other replicas of the system aside from the primary, analogous to the lieutenants in the simplified problem.

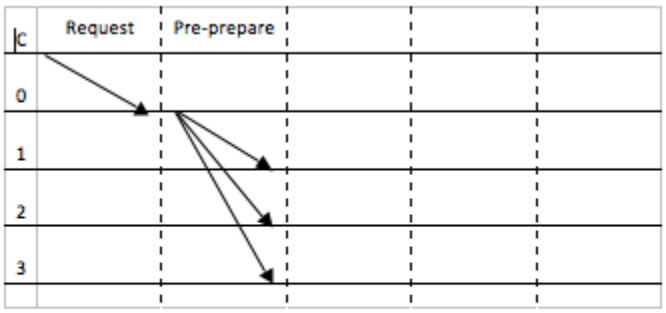
Each process occurs under what is called a *view* which is nothing more than a record of which replica has been designated as the primary replica. Each different view would therefore correspond to a different replica being designated as the primary. The view will take on a more important role later when we consider a system with a faulty primary replica. A formal proof of the correctness of the algorithm is beyond the scope of this lecture, but interested readers should refer to "*A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm*" [7]. What follows is an outline of the process and then each part will be expanded and the details filled in.

During the first round of the protocol the client makes a request of the system. This request comprises the action or operation that the system is to perform.



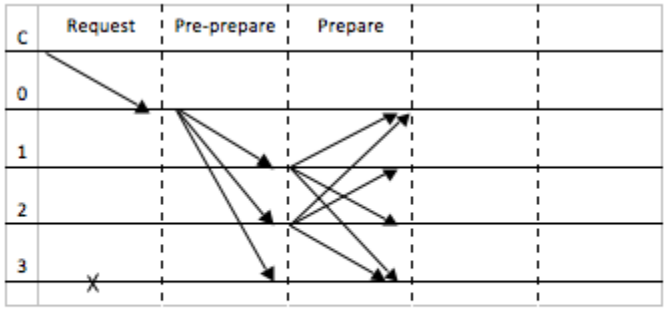
Normal Case Operation
C is the client, 0 the primary, and 3 is the faulty node

During the second round, called the *Pre-Prepare* phase, the primary replica forwards the request to all other replicas in the system.



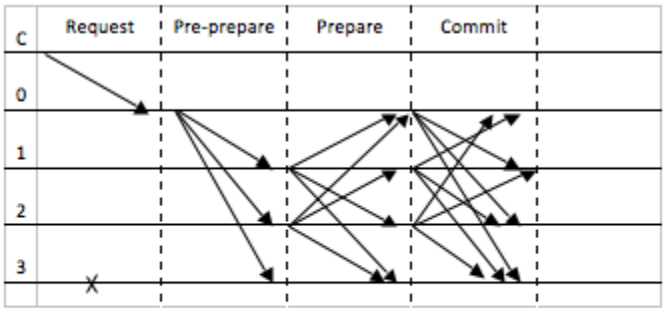
Normal Case Operation
C is the client, 0 the primary, and 3 is the faulty node

During the third round, called the *Prepare* phase, the backup replicas send a message to all other replicas, including the primary, stating what the request was that they had received. Some replicas, if faulty, may not send a message to the others. In this example replica 3 is faulty and does not send a message.



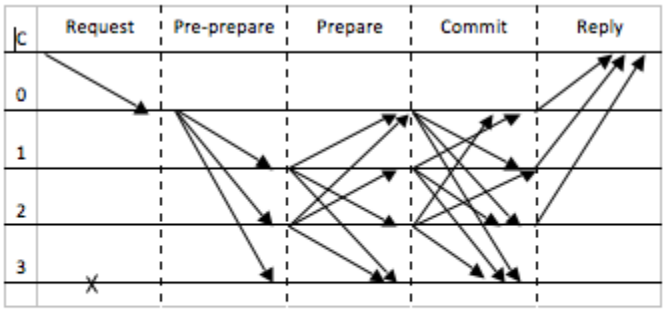
Normal Case Operation
C is the client, 0 the primary, and 3 is the faulty node

During the fourth round, if the replicas receive a quorum, i.e. a majority of messages from the other backup replicas that match the initial message sent in the *Pre-Prepare* phase by the primary, then the replicas will send out a *Commit* message to each of the other replicas.



Normal Case Operation
C is the client, 0 the primary, and 3 is the faulty node

Finally if the replicas receive a majority of *commit* messages that match, then each replica will send a reply to the client. The reply comprises the completed operation or computation that was initially requested.



Normal Case Operation
C is the client, 0 the primary, and 3 is the faulty node

2.3 Details of Algorithm

During each round of the protocol, messages are passed that contain specific information that helps identify the sender and validate the message itself.

1. **Round 1:** The client sends a request to the primary node in the form $\langle \text{REQUEST}, o, t, c \rangle_{\sigma C}$. Here the o is the operation or computation that the system as a whole is to perform. The results of this operation are what is being sent back to the client at the end. The t is the timestamp of when the request was sent (e.g. the clock time off the client's machine). Since the system is operating in asynchrony each different replica may be in various states of processing many different requests and the timestamp helps the replicas prioritize the operations to be performed. The c is designates from which client

the request originated. The σc is the signature of the client and is used under the signed message model to ensure that the request has not been forged.

2. **Round 2:** The primary replica assigns a value n to the message it receives from the client and creates a *pre-prepare* message in the form $\langle\langle$PRE-PREPARE, *v, n,* $d\rangle_{\sigma p}$, $m\rangle$. The current view designating which replica is the primary replica is denoted by v. The n is the number assigned to the request from the client, which is used like the timestamp to order the requests being processed as well as for validation purposes. The message digest is denoted as d and is part of the signed messaging model. It is compared against m, which is the client's request message, to ensure the validity of the original request. The σp is the signature of the primary and is used to ensure that the request has not been forged by the primary. The other replicas accept this *pre-prepare* message from the primary and store it in their log if:

   a. The signatures in the request and pre-prepare messages are correct and d is the digest for m.

   b. It is in view v

   c. It has not already accepted another message under view v with message number n and a different digest.

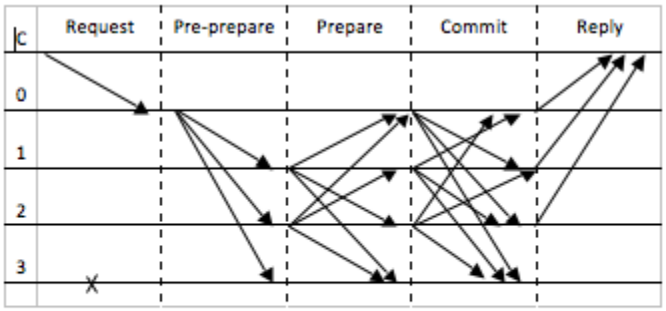
   d. The number n is within an acceptable range (a malicious primary may try to exhaust the sequence numbers by assigning a very large value for n.)

3. **Round 3:** If the backup replicas accept the *pre-prepare* message from the primary, they multicast their own message to all other nodes in the form $\langle$PREPARE, *v, n, d, i*$\rangle_{\sigma i}$. The v, n, and d are the view, message number, and message digest respectively as described above. The i signifies the specific backup replica sending the message, and the message is signed as before with the σi representing the signature of the individual replica. If the backups do not accept the message from the primary because one of the conditions above is not met, then they do nothing. All $\langle$PREPARE, *v, n, d, i*$\rangle_{\sigma i}$ messages are added to the logs of each replica if they have valid values. Messages are validated, or *prepared*, if replica i has in its log a *pre-prepare* for the original message from the client m originally sent by the primary and $2f$ *prepares* from backup replicas that match the *pre-prepare*.

4. **Round 4:** When a message is *prepared*, replica i sends a *commit* message to all other nodes in the form $\langle$COMMIT, *v, n, d, i*$\rangle_{\sigma i}$, where the values are the same as in round 3. All nodes add the *commit* messages to their logs if they have valid values. Finally a message is *committed-local (m, v, n, i)* if and only if *prepared(m, v, n, i)* is true and i has accepted $2f + 1$ commits (including its own) from other replicas that match the *pre-prepare* for m. The $2f + 1$ is significant because it represents the minimum number of non-faulty replicas.

Since roughly one third of the replicas could be faulty, the $2f$ represents the two thirds that are guaranteed to be functional.

5. **Round 5:** The backup nodes execute the request and send a reply to the client once *committed-local (m, v, n, i)* is true. The client waits for $f + 1$ replies that are the same, where f is the maximum number of faulty nodes allowable based on the restriction that at most $\lfloor \frac{n-1}{3} \rfloor$ out of the total of n nodes are faulty. The reply messages have the form $\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma i}$ where v is the current view signifying the primary node, t is the timestamp of the request, c is the client the message is sent to, i is the replica number, and r is the result of executing the operation o that was initially requested from the system. Waiting for $f + 1$ matching results guarantees that the reply is valid since there can be at most f faulty nodes.



Normal Case Operation
C is the client, 0 the primary, and 3 is the faulty node

2.4 When Things Go Wrong

Failures can either occur with the primary replica or with the backup replicas. For this algorithm it is assumed that we can guard against faulty clients using some form of authentication and access control that prohibits clients from making invalid requests or invoking invalid operations.

1. **Primary:** The primary replica could default by failing to pass along a request or sending conflicting *pre-prepare* messages. In the first case, the system will time out, no replies will be sent to the client, and the client will move to a secondary request procedure. Under the secondary request procedure the client's request is resubmitted individually to all the backup replicas. The backup replicas will then forward the request to the current primary and wait for a multicast broadcast from the primary. In the event that they do not receive a *pre-prepare* message, enough backup replicas will suspect a faulty primary and cause a view change.

   The view-change protocol allows the system to progress in the presence of a faulty primary replica. When the backup replicas send out a *view-change*

message requesting the next view $v + 1$, the primary replica *p'* designated as the primary for view $v + 1$ waits for $2f$ valid-change messages for view $v + 1$ before sending out a next-view message to all backup replicas. The backup replicas then update their view number and a message is sent to the client to update the client's view. During the view change a backup of all requests is created that allows the system to continue from where it was before the failure of the primary.

In the second case in which the primary sends out conflicting *pre-prepare* messages, the backup replicas will fail to reach a consensus of $2f$ *prepare* messages that match the *pre-prepare* and will do nothing. Again the system will time out and cause a view change.

Any fault from the primary will cause a view change and will move the system to the next primary. This ensures that primary replica failures will never terminate the process

2. **Backups:** The failures of the backups are already accounted for in the algorithm itself assuming that at most $\lfloor \frac{n-1}{3} \rfloor$ out of the total of n nodes are faulty.

2.5 Analysis

The analysis of a distributed system is a little different from the analysis of the runtime on a single CPU. In the distributed case the atomic operations that are measured for the runtime analysis are the messages passed rather than the particular action performed by the individual replicas. This is because the runtime is dictated by the message passing over the network since things travel more slowly across a network than within a single CPU. By counting the number of messages passed an accurate view of the asymptotic running time of the system can be achieved.

In round 1 a single message is passed between the client and the primary and the time is $O(1)$. In the second round messages are passed from the primary to all backup replicas for a total of $(n - 1)$ messages or $O(n)$ time. In the third and fourth round messages are passed from all replicas to $n - 1$ other replicas for a total of $(n)(n - 1)$ messages or $O(n^2)$ time. So overall the time is dominated by the $O(n^2)$ term. However, in the worst case the faulty replicas would all line up to be primaries and the time for view changes would have to be accounted for. For each view change there are two rounds of $(n)(n - 1)$ messages being sent, or $O(n^2)$ time. But if all the faulty replicas lined up and there were a maximum number of faulty replicas then the time would be the $O(n^2)$ for the message passing $\frac{n}{3}$ times for an overall time of $O(n^3)$.

2.6 Optimizations

1. **Reducing Replies:** Since all backup replicas send a reply to the client, large reply messages can tax the bandwidth and significantly slow down the protocol. In one optimization the client designates a single replica to send the reply message. The designated replica sends the full reply message and the other

replicas send only the message digest. The digests from all replicas are used to verify the validity of the single message. If the message is valid it is accepted, otherwise the client retransmits the original request and specifies that a full message response should be sent from all replicas.

2. **Tentative Replies:** Under normal operation the replicas all commit before sending a reply to the client. With one optimization the replicas will send a tentative reply to the client before committing. If the client receives $2f + 1$ matching tentative replies it means the message will eventually commit and the client can proceed with the tentative reply. If it does not receive $2f + 1$ matching tentative replies, then the client resubmits its request and waits for $f + 1$ none tentative replies. This effectively takes out one round of message passing among the replicas.

3. **Cryptography:** One of the major performance bottlenecks of digital signature algorithms is the computations involved when using digital signatures to authenticate all messages [8]. Another optimization of the Practical Byzantine Protocol uses digital signatures only during view changes for *view-change* and *new-view* messages. Otherwise it uses message authentication codes which are three orders of magnitude faster. A message authentication code is a secret session key that is shared by each node, including clients, and each replica.

References

[1] M. Pease, R. Shostak, L. Lamport (April 1980). *"Reaching agreement in the presence of faults"*. J. ACM 27 (2): 228234. http://doi.acm.org/10.1145/322186.322188.

[2] Amazon Web Services. http://status.aws.amazon.com/s3-20080720.html

[3] M. Pease, R. Shostak, L. Lamport. ACM Transactions on Programming Languages and Systems, July 1982, pages 382-401 http://pages.cs.wisc.edu/ sschang/OS-Qual/reliability/byzantine.htm

[4] M. Pease, R. Shostak, L. Lamport. *"The Byzantine Generals Problem"*. http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf

[5] A. Bar-Noy, D. Dolev, C. Dwork, H. Strong. *"Shifting Gears: Changing Algorithms on the Fly to Expediate Byzantine Agreement"*. http://www.cs.huji.ac.il/ dolev/pubs/shifting-gear.pdf

[6] M. Fischer, N. Lynch, M. Paterson. *"Impossibility of Distributed Consensus with One Faulty Process"*. Second ACM Symposium on Principles of Database Systems, March 1983. Retrieved from http://groups.csail.mit.edu/tds/papers/Lynch/pods83-flp.pdf

[7] M. Castro, B. Liskov. *"A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm"*. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, June 1999. Retrieved from http://pmg.csail.mit.edu/ castro/tm590.pdf

[8] M. Castro, B. Liskov. *"Practical Byzantine Fault Tolerance"*. Third Symposium on Operating Systems Design and Implementation, February 1999. Retrieved from http://research.microsoft.com/en-us/um/people/mcastro/publications/osdi99.pdf