Mark Nelson

Programming, mostly.

Search

- Home
- About Mark Nelson
- Archives
- Liberal Code Use Policy

The Byzantine Generals Problem

« Abraham Lempel Honored by IEEE
Phishers take it to the next level »

Posted in July 23rd, 2007
by Mark Nelson in Computer Science, Magazine Articles, Programming

This article presents the algorithm that solves the Byzantine General's Problem, as first described by Lamport, Pease, and Shostak in 1982 [1]. While Lamport's algorithm is not particularly complex, programmers who aren't used to working on distributed computation might find it difficult to implement. To accompany the explanation of the algorithm, I have included a C++ program designed for experimentation with the solution.

Introduction

The Byzantine General's Problem is one of many in the field of agreement protocols. In 1982, Leslie Lamport described this problem in a paper written with Marshall Pease and Robert Shostak. Lamport framed his paper around a story problem after observing what he felt was an inordinate amount of attention received by Dijkstra's Dining Philosophers problem [2].

This problem is built around an imaginary General who makes a decision to attack or retreat, and must communicate the decision to his lieutenants. A given number of these actors are traitors (possibly including the General.) Traitors cannot be relied upon to properly communicate orders; worse yet, they may actively alter messages in an attempt to subvert the process.

When we're not dwelling in storybook land, the generals are collectively known as *processes*, the general who initiates the order is the *source process*, and the orders sent to the other processes are *messages*. Traitorous generals and lieutenants are *faulty processes*, and loyal generals and lieutenants are *correct processes*. The order to retreat or attack is a message with a single bit of information: a one or a zero.

In general, a solution to an agreement problem must pass three tests: *termination*, *agreement*, and *validity*. As applied to the Byzantine General's problem, these three tests are:

1. A solution has to guarantee that all correct processes eventually reach a decision regarding the value of the order they have been given.
2. All correct processes have to decide on the same value of the order they have been given.
3. If the source process is a correct process, all processes have to decide on the value that was original given by the source process.

Note that one interesting side effect of this is that if the source process is faulty, all other processes still have to agree on the same value. It doesn't matter what value they agree on, they simply all have to agree. So if the General is subversive, all lieutenants still have to come to a common, unanimous decision.

Difficulties

This agreement problem doesn't lend itself to an easy naïve solution. Imagine, for example, that the source process is the only faulty process. It tells half the processes that the value of their order is zero, and the other half that their value is one.

After receiving the order from the source process, the remaining processes have to agree on a value that they will all decide on. The processes could quickly poll one another to see what value they received from the source process.

In this scenario, imagine the decision algorithm of a process which receives an initial message of zero from the source process, but sees that one of the other processes says that the correct value is one. Given the conflict, the process knows that either the source process is faulty, having given different values to two different peers, or the peer is faulty, and is lying about the value it received from the source process.

It's fine to reach the conclusion that someone is lying, but making a final decision on who is the traitor seems to be an insurmountable problem. And in fact it can be proven that it is impossible to decide in some cases. The classic example used to show this is when there are only three processes: one source process and two peer processes.

In the two configurations shown in Figure 1 and Figure 2, the peer processes attempt to reach consensus by sending each other their proposed value after receiving it from the source process. In Figure 1, the source process (P_1) is faulty, sending two different values to the peers. In Figure 2, P_3 is faulty, sending an incorrect value to the peer.

You can see the difficulty P_2 faces in this situation. Regardless of which configuration it is in, the incoming data is the same. He has no way to distinguish between the two configurations, and no way to know which of the two other processes to trust.

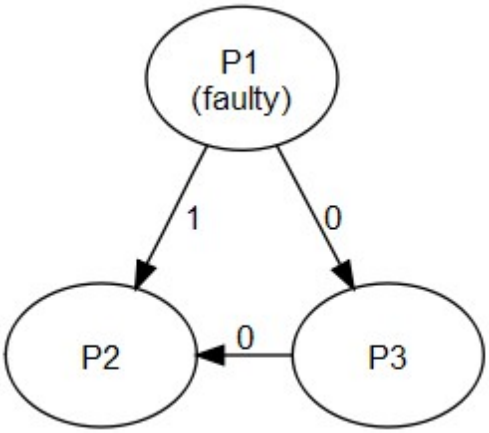

Figure 1
The case in which the source process is faulty

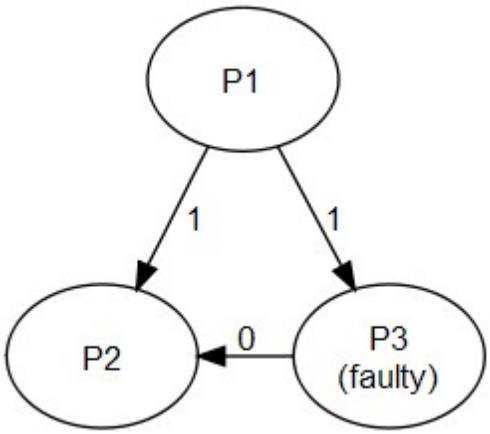

Figure 2
The case in which P_3 is faulty

This situation doesn't necessarily get better just by throwing more non-faulty processes at the problem. A naïve algorithm as shown in Figure 1 and Figure 2 might have each process tell every other process what it received from P_1. A process would then decide on the correct value by taking a simple majority of the values in its incoming messages.

Under the rules of this approach, it is easy to show that regardless of how many processes are in the system, a subversive source process with one collaborator can cause half the processes to choose to attack, while half the processes elect to retreat, leading to maximum confusion.

The Lamport, Pease and Shostak Algorithm

In 1982, Lamport, Pease, and Shostak published a fairly simple solution to this problem. The algorithm assumes that there are *n* processes, with *m* faulty processes, where $n > 3m$. Thus, for a scenario such as that in Figure 1 and 2 with 1 faulty process, there would have to be a minimum of 4 processes in the system to come to agreement. (For the rest of this article, *n* will always refer to the count of processes, and *m* will always refer to the number of faulty processes.)

The definition of the algorithm in the original paper is short and succinct, but at least in my experience, is somewhat confusing for programmers without a lot of experience in distributed algorithms.

Lamport's algorithm is a recursive definition, with a base case for *m=0*, and a recursive step for m > *0*:

Algorithm OM(0)

1. The general sends his value to every lieutenant.
2. Each lieutenant uses the value he receives from the general.

Algorithm OM(m), m > 0

1. The general sends his value to each lieutenant.
2. For each i, let v_i be the value lieutenant i receives from the general. Lieutenant i acts as the general in Algorithm OM(m-1) to send the value v_i to each of the n-2 other lieutenants.
3. For each i, and each $j \neq i$, let v_i be the value lieutenant i received from lieutenant j in step 2 (using Algorithm (m-1)). Lieutenant i uses the value majority (v_1, v_2, … v_n).

Lamport's Algorithm Definition

To most programmers, this is going to look like a conventional recursive function definition, but it doesn't quite fit into the mold you learned when studying the example of *factorial(n)*.

Lamport's algorithm actually works in two stages. In the first step, the processes iterate through *m+1* rounds of sending and receiving messages. In the second stage of the algorithm, each process takes all the information it has been given and uses it to come up with its decision.

I found this to be non-obvious without quite a bit of study, which is the reason for this article.

The First Stage

The first stage of the algorithm is simply one of data gathering. The algorithm defines *m+1* rounds of messaging between all the processes.

In round 0, the General sends the order to all of its lieutenants. Having completed his work, the General now retires and stands by idly waiting for the remaining work to complete. Nobody sends any additional messages to the General, and the General won't send any more messages.

In each of the remaining rounds, each lieutenant composes a batch of messages, each of which is a tuple containing a value and a path. The value is simply a 1 or a 0. The path is a string of process ids, <ID_1, ID_2, …, ID_n>. What the path means in this context is that in Round N, PID_N is saying that was told in round N-1 that P_{IDN-1} was told by… P_{ID1} that the command value was *v*. (This is very much like the classic party game in which a message is whispered from ear to ear through a chain of players, becoming slightly mangled along the way.) No path can contain a cycle. In other words, if ID1 is 1, no other ID in the string of process IDs will be a 1.

The message definition is easy in round 1. Each process broadcasts a message to all the other processes, including itself, but excluding the General, with the value it received from the General and its own process ID.

In subsequent rounds, things get more complicated. Each process takes all the messages it received from the previous round, appends its process ID where allowed, and sends those messages to all other processes, including itself. (The "where allowed" just means that the process skips any messages where adding its process ID to the list would create a cycle in the string of process IDs.)

For example, let's suppose that in Round 0 that P_1, a faulty general told P_2, P_3, and P_4 that the command value was 0, and told P_5, P_6, and P_7 that the command value was 1. In round 1, the following messages would be sent:

Sender=P_2		Sender=P_3		Sender=P_4		Sender=P_5		Sender=P_6		Sender=P_7	
Dest	Msg	Dest	Msg	Dest	Msg	Dest	Msg	Dest	Msg	Dest	Msg
P_2	{0,12}	P_2	{0,13}	P_2	{0,14}	P_2	{1,15}	P_2	{1,16}	P_2	{1,17}
P_3	{0,12}	P_3	{0,13}	P_3	{0,14}	P_3	{1,15}	P_3	{1,16}	P_3	{1,17}
P_4	{0,12}	P_4	{0,13}	P_4	{0,14}	P_4	{1,15}	P_4	{1,16}	P_4	{1,17}
P_5	{0,12}	P_5	{0,13}	P_5	{0,14}	P_5	{1,15}	P_5	{1,16}	P_5	{1,17}
P_6	{0,12}	P_6	{0,13}	P_6	{0,14}	P_6	{1,15}	P_6	{1,16}	P_6	{1,17}
P_7	{0,12}	P_7	{0,13}	P_7	{0,14}	P_7	{1,15}	P_7	{1,16}	P_7	{1,17}

Table 1
Messages sent by all six lieutenant processes in round 1

The number of messages goes up in in the second round. From the previous iteration, we know that each process now has six values that it received in the previous round – one message from each of the six other non-source processes – and it needs to send each of those messages to all of the other processes, which might mean each process would send 36 messages out.

In the previous table I showed the messages being sent to all six processes, which is fairly redundant, since the same messages are broadcast to all processes. For round 2, I'll just show you the set of messages that each process sends to all of its neighbors.

Sender=P_2	Sender=P_3	Sender=P_4	Sender=P_5	Sender=P_6	Sender=P_7
{0,132}	{0,123}	{0,124}	{0,125}	{0,126}	{0,127}
{0,142}	{0,143}	{0,134}	{0,135}	{0,136}	{0,137}
{1,152}	{1,153}	{1,154}	{0,145}	{0,146}	{0,147}
{1,162}	{1,163}	{1,164}	{1,165}	{1,156}	{1,157}
{1,172}	{1,173}	{1,174}	{1,175}	{1,176}	{1,167}

Table 2
Messages sent by all six processes in round 2

The six messages that P_2 received in round 1 were {0,12}, {0,13}, {0,14}, {1,15}, {1,16}, and {1,17}.

According to the earlier definition, P_2 will append its process ID to the path and forward each resulting message to all other processes. The possible messages it could broadcast in round 2 are {0,122}, {0,132}, {0,142}, {1,152}, {1,162}, and {1,172}. The first message, {1,122} contains a cycle in the path value of the tuple, so it is tossed out, leaving five messages to be sent to all processes.

The first message that P_2 is sending in round 2, {0,132}, is equivalent to saying "P_2 is telling you that in round 1 P_3 told it that in round 0 that P_1 (the General) told it that the value was 0″. The five messages shown in P_2's column in the table are sent to all six lieutenant processes, include itself.

It's easy to see that as the number of processes increases, the number of messages being exchanged starts to go up rapidly. If there are N processes, each process sends N-1 messages in round 1, then (N-1)*(N-2) in round 2, (N-1)*(N-2)*(N-3) in round 3. That can add up to a lot of messages in a big system.

The Second Stage

While sending messages in each round, processes are also accumulating incoming messages. The messages are stored in a tree format, with each round of messages occupying one rank of the tree. Figure 3 shows the layout of the tree for a simple configuration with six processes, one of which can be faulty. Since m=1, there are just two rounds of messaging: the first, in which the general sends a value to each lieutenant process, and a second, in which each process broadcasts its value to all the other processes. Two rounds of messaging are equivalent to two ranks in the tree.

Each node in the tree has three elements: an input value, a path, and an output value. The input value and path are defined in the first stage of the algorithm – they are simply the messages received from the peer processes. The output value is left undetermined until the second stage of the algorithm, which I am defining here. Note that in the figure below, the output values are initially set to '?', indicating that they are presently unknown.

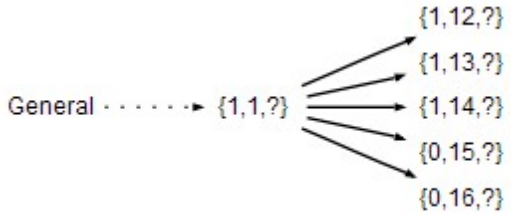

Figure 3
The Tree Layout for 5 processes with 1 faulty process

In Figure 3, there are six processes, and the General (P_1) is faulty – sending a 1 to the first three lieutenants and 0 to the last two. The subsequent round of messaging results in P_2 having an information tree that looks just like that shown in Figure 3. (Because only the General is faulty, in this case all other processes will have an identical tree.)

Once a process has completed building its tree, it is ready to decide on a value. It does this by working its way up from the leaves of the tree, calculating the majority value at each rank and assigning it to the rank above it. The output value at each level is the third item in the data structure attached to each node, and those values are all undefined during the information gathering stage.

Calculating the output values is a three step process:

1. Each leaf node in the tree (all values at rank m) copies its input value to the output value.
2. Starting at rank m-1 and working down to 0, the output value of each internal node is set to be the majority of the output values of all its children. In the event of a tie, an arbitrary tie-breaker is used to assign a default value. The same default value must be used by all processes.
3. When complete, the process has a decision value in the output of the sole node at rank 0.

In Figure 3, step 1 of the process assigns the initial values to the leaf nodes. In the next step, the majority value of { 1, 1, 1, 0, 0 } is evaluated and returns a value of 1, which is assigned to the output value in rank 0. Because that is the top rank, the process is done, and P_1 decides on a value of 1.

Every lieutenant value in a given exercise will have the same paths for all its nodes, and in this case, since only the General is faulty, we know that all lieutenants will have the same input values on all its leaves. As a result, all processes will agree on the same value, 1, which fulfills the agreement property.

A More Complicated Example

Getting a good understanding of the algorithm really requires walking through an example that has at least three ranks. (Examples on the web, mostly extracted from lecture notes, nearly always have the simple two-rank example.) For this example, consider an example with *n=7* and *m=2*. We'll continue with the convention that the General is P_1, and instead of having a faulty general, we'll have P_6 and P_7 be faulty processes. After the initial three rounds of information exchange, we have the three-ranked tree shown in Figure 4:

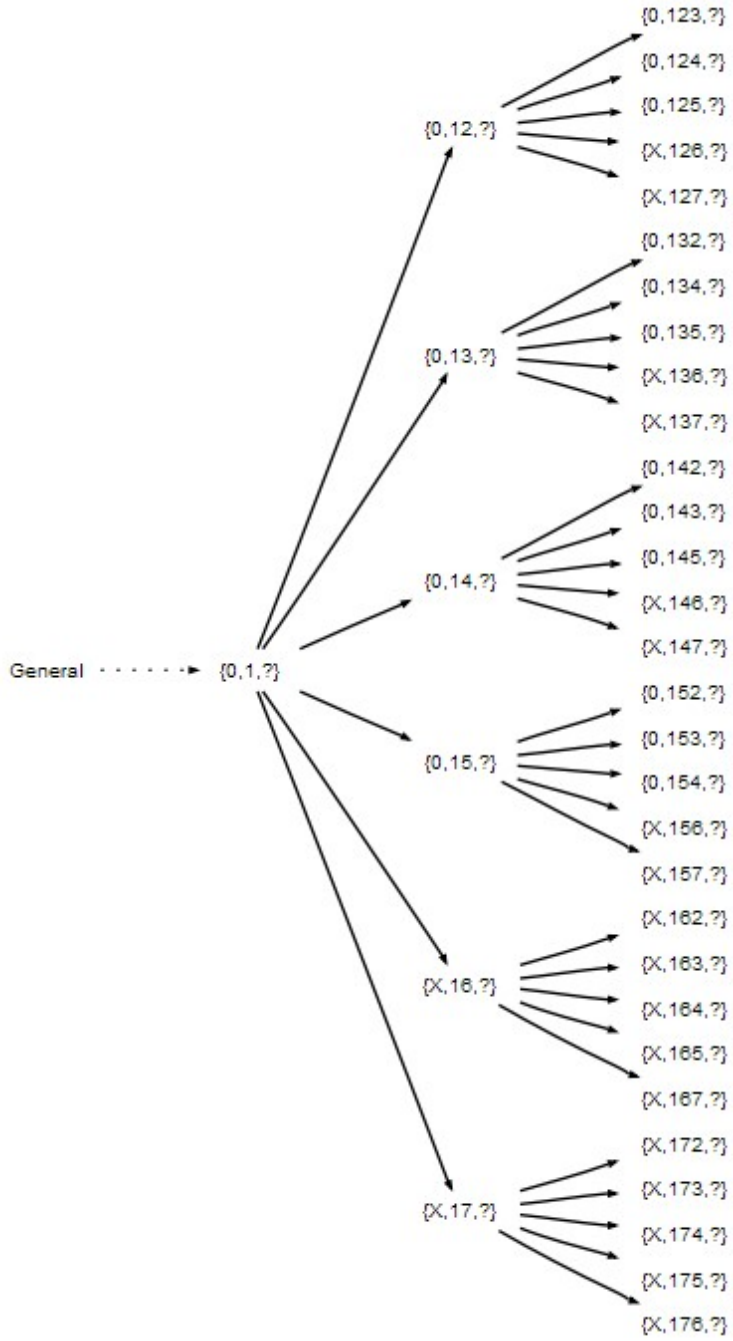

Figure 4
A tree with $n=7$, $m=2$, and faulty processes P_6 and P_7

The important thing to note in these trees is that I've inserted the value 'X' for the input values of any input value that comes from the two faulty processes. We don't know what P_6 and P_7 might send in any given round, so in general, we'll try to work through the algorithm without constricting their incorrect messages to any specific values.

You'll see that at rank 1, the values from path 17 and 16 are both set to X. In the first round the two faulty processes communicated possibly false values to all other processes, and may have arbitrarily changed the values sent to different processes in order to skew the results.

As a result of those bad values in rank 1, we see their frequent occurrence in rank 2. The incorrect values show up not only in direct messages from the faulty processes, but also in any message from a correct process

that includes a faulty process earlier in its path.

All in all, at the leaf nodes, we have 18 deceptive values at the leaf nodes, and only 12 accurate messages that trace their way all the way back to the general through nothing but correct processes. Obviously, if we just voted on the majority of the messages we had received, we would be susceptible to falling for the wrong value.

Fortunately, the layout of the tree guarantees that we will actually get a correct value. In Figure 4, the roll up of the output values hasn't occurred yet, so every node has a question mark in the output value. In Figure 5, the output values are shown. The leaf rank has the output values set to the input values, with X used to indicate unknown values from faulty processes.

When the leaf rank is rolled up to the second rank, the nodes with paths 12, 13, 14, and 15 all have clear majority values of 0 for their output values, with 16 and 17 set to X, as their values are uncertain.

The final roll up to the top rank successfully sets the output value to 0, as four of the inputs are set to 0 and only 2 are set to X. Mission accomplished. And because of the way this was calculated, we know that the correct result will be achieved regardless of what deceptive values are sent by the two faulty processes.

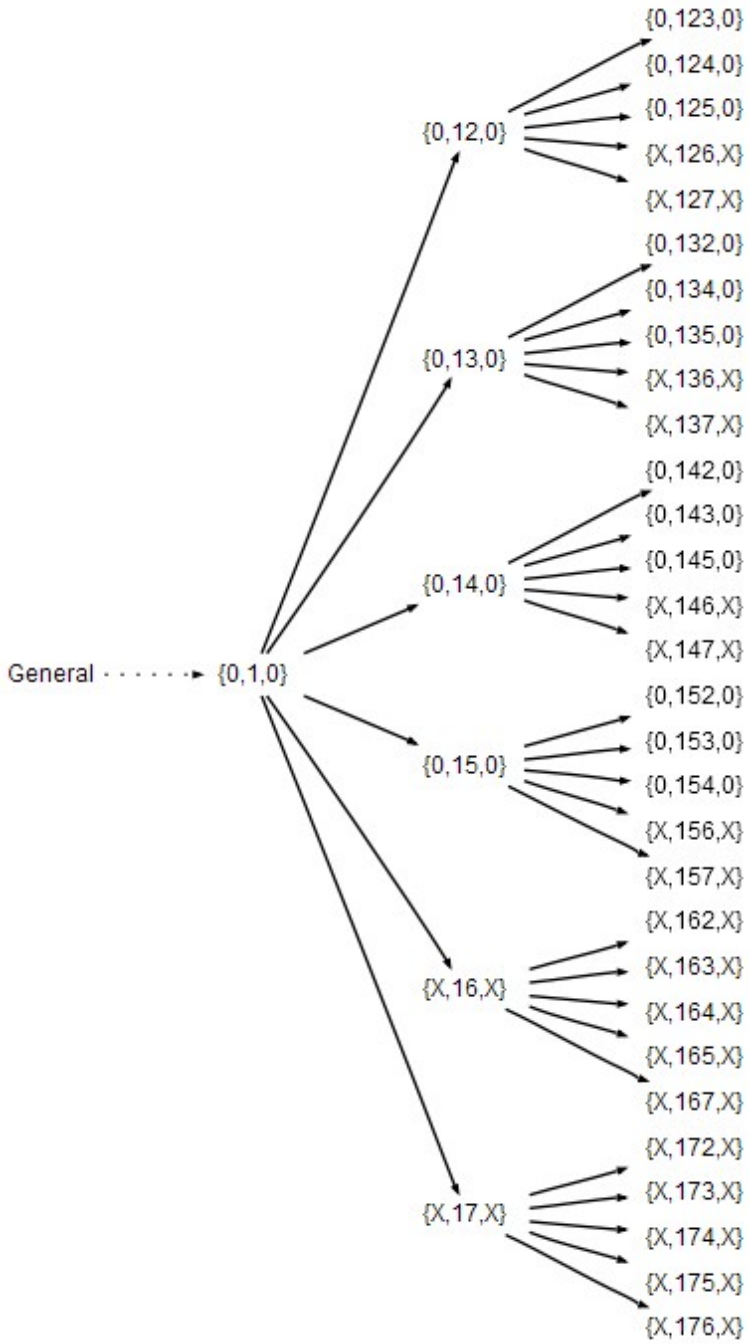

Figure 5
The tree after calculating the output values

The Sample Code

I've included a simple C++ program that implements this algorithm, with extensive internal documentation. It has a `Process` class that is used to send and receive messages, as well as to roll up the decision tree. A `Traits` class is used to define the number of processes, the number of faulty processes, the source process, and what values the faulty processes send in various rounds.

To help with visualization, the program will output the tree for a given process in the format used by *dot*, part of the free Graphviz program. You can then use dot to create a nice picture of the output graph – all the figures in this article were created that way. I find that using the SVG format for the output produces very readable results.

As supplied, the program is set for values of *n=7* and *m=2*. Good exercises to perform while experimenting with it include:

- Attempt to invalidate the program or the algorithm by getting incorrect results with some particular combination of faulty messages.
- Add a third faulty process and show that it is relatively easy to get invalid output when *n=7* and *m=2*.
- Reduce *n* to 6 and show that it is relatively easy to get invalid output with two faulty processes.
- Move up to *m=3* and *n=10*. Experiment with various combinations of faulty Generals and lieutenants and see if you can create incorrect results.

Note

Most implementations of this algorithm include logic designed to deal with the case in which a faulty process fails to send a message. This is equivalent to simply having the faulty process send an arbitrary value, so I don't treat it as a separate case.

Source Code

source.zip, which contains:

- main.cpp
- README.TXT
- VS2003/byzantine.sln
-
- VS2003/byzantine.vcproj
- VS2005/byzantine.sln
-
- VS2005/byzantine.vcproj

References

[1] The Byzantine Generals Problem (with Marshall Pease and Robert Shostak)
ACM Transactions on Programming Languages and Systems 4, 3 (July 1982), 382-401.
http://research.microsoft.com/users/lamport/pubs/byz.pdf

[2] The Writings of Leslie Lamport: http://research.microsoft.com/users/lamport/pubs/pubs.html#byz

Lynch, Nancy A. Distributed Algorithms. San Francisco, CA: Morgan Kaufmann, 1997. ISBN: 1558603484.

Graphviz – Graph Visualization Software. http://www.graphviz.org/

CS 6378: Advanced Operating Systems Section 081 Notes on Agreement Protocols (lecture notes by Neeraj Mittal, University of Texas at Dallas) http://www.utdallas.edu/~neerajm/cs6378su07/agreement.pdf

Update 01-November-2007
Distributed Computing With Malicious Processors w/o Crypto or Private Channels, YouTube Video of Google TechTalk, Valerie King, Department of Computer Science, University of Victoria, Victoria, BC, Canada

30 users commented in " The Byzantine Generals Problem "

Follow-up comment rss or Leave a Trackback

on August 30th, 2007 at 9:03 pm, Shahzad Bhatti said:

I enjoyed this interesting problem and your solution, but I noticed that in your source code on line 400, you are switching n and m values. Also, though my C is rusty, but shouldn't ids vector on the same method (GenerateChildren) be passed by reference.

on August 31st, 2007 at 4:53 am, Mark said:

Hi Shahzad,

Switching n and m looks like a bug to me.. I'm surprised the program worked properly in that state. Perhaps I just got lucky. I'll try to get it properly diagnosed and fixed today.

As for passing the vector by reference, yes, that would probably be a bit more efficient, but the vector is pretty small, and GenerateChildren() is only called once at startup, so I'm not really worried about it.

on September 4th, 2007 at 9:31 am, Hippopottoman said:

Hi.

I enjoyed the article very much, but if I read correctly, there's a discrepency between the definition of Algorithm OM(m), m > 0 and Figure 1 (and the paragraphs immediately following the figure).
P2 shouldn't be sending messages to itself in Figure 1, only to P3, P4, P5, P6, and P7. Similarly for the other Pis, i in {3, 4, 5, 6, 7}.

on September 4th, 2007 at 10:24 am, Mark said:

>but if I read correctly, there's a discrepency between the definition
>of Algorithm OM(m), m > 0 and Figure 1

Yes, I agree that it appears that way.

However, if you actually look at the diagrams in Lamport's paper, what you see is that each lieutenant collects votes from all his peers, and from the general in order to make his decision.

That vote that comes from the general is actually the same value that Pn sends to itself – it is a convenience to do it this way.

So think of the message that Pn sends to itself as simply being the value that it received from the general one round up, and it will all make sense.

on September 5th, 2007 at 1:48 pm, Shahzad Bhatti » Blog Archive » Performance testing C++, Java, Ruby and Erlang said:

[...] I came across an interesting distributed problem Byzantine Generals Problem from Mark Nelsons' blog. He showed C++ implementation of Leslie Lamport algorithm. It seemed like a natural fit Erlang, but writing directly in ErlangI wrote the algorithm into Java and Ruby (with slight redesign). Unfortunately, the original C++ source, and my Java and Ruby versions are not really truly distributed or concurrent. I just wanted to translate the original algorithm without changing a whole a lot, but Erlang gave you both distributing features and concurrency for free. [...]

on October 21st, 2008 at 5:54 am, Vis said:

Hi,
your very useful explanation of the Byzantine Problem leave me a doubt.
When you say that P(0,132) means that P2 is telling you that in round 0 P1 told it tat the value was 0,
if general 2 send this message to general 3 (he should broadcast his message with path to all other processes),
does it make sense(or is it useful to reach ICI) the sentence
"I say you that you said me that 1 told you this value"?
Every general must build the whole tree or he can prune his own branch, not receiving cicling messages?

Thanks,

Roberto

 on October 21st, 2008 at 8:17 am, Mark Nelson said:

@Vis:

Every general has to build the tree because every tree may be different – depending on faulty messages.

I don't understand the first part of your question though.

on October 22nd, 2008 at 4:20 am, VIs said:

I mean,

when a loyal general,for example 2 , broadcast his message after appending his ID i.e. P(12),
the other general receaving this message append their ID and broadcast the message.
Is it useful that the loyal general 2 receive again a message he sent before?
I think that in the recursive implementation each lieutenant when receive a message, perform as a general in algorithm
OM(m-1),so he send his message and hold. He does not receive back his own value again processed by other lieutenants.

Roberto

 on October 22nd, 2008 at 6:40 am, Mark Nelson said:

@VIs:

For the algorithm to work, each general has to broadcast each message to all other generals, including himself. Your idea of not broadcasting messages to someone who has already seen it will lead to possible errors – I think you should be able to construct such a scenario pretty quickly.

on November 26th, 2008 at 2:35 pm, Paul said:

Hi Mark,

Couple of questions. I haven't looked through your code.

1. Does each process know many other other processes there are (the 'n' number)?
2. If so, the algorithm needs to go through m+1 rounds… how do the processes know in advance how many faulty processes there are? Or does the (actually) alogorithm go through n rounds of messaging, but it turns out everything is deterministic after m+1 rounds?

Thanks,

Paul

 on November 26th, 2008 at 2:49 pm, Mark Nelson said:

@Paul:

yes each process has to know how many other processes exist so it can create a message for each one.

Nobody knows how many faulty processes exist. The value of m is chosen to ensure validity, but if the number of errant processes is too large, the algorithm won't work. If the number is small, the algorithm is wasting time, but still works properly.

on November 26th, 2008 at 3:43 pm, Paul said:

Thanks for the reply, Mark.

If none of the processes knows how many errant processes there are, then how does each process know how many rounds 'm+1 rounds' is? That's my main question, really. We can stipulate it in the prescient role of programmer, but in real life how would that play out?

Paul

 on November 27th, 2008 at 12:08 pm, Mark Nelson said:

@Paul:

You're missing the point here.

You design a system such that it can tolerate a certain amount of error.

That's all there is too it.

If you have a space shuttle with 6 computers, you might design a system that can tolerate 2 failures. You pick your m and n according to these rules.

If three systems fail, the shuttle crashes.

on December 1st, 2008 at 1:07 pm, Paul said:

Mark, you're right. I wasn't thinking in terms of design, rather in terms of maintenance and error avoidance in an operating environment.

I did my own implementation and had a question. Let's say we have 7 generals including 2 traitors, one of which is the commander. Therefore, I had the commander randomly generate a "X" or a 0 to send to the other N-1 generals. If enough X's are sent out by the commander, then the majority voting could turn out to have a predominance of X's at the end of the process. Yet, an X could mean anything: non-existence of a message or any random screwed up message (ie, a Y,Z,B,D,… not just an X). So I would imagine each loyal general that calc's an X as the majority would say "Hmmm… the commander is a traitor, or there are too many traitors in our midst. I have to go with the default order (of retreat or attack"). Is this the correct way to interpret the algorithm in your opinion?

Thanks,

Paul

 on December 1st, 2008 at 2:52 pm, Mark Nelson said:

@Paul:

No, the lieutenants don't have any way of knowing that a command is good. If you have a commander sending out bad commands, all we know for sure after the algorithm runs is that all the honest lieutenants will arrive at the same command. They won't actually know if it is from an honest leader or not. At least they will all do the same thing.

We also know that if the leader is honest, all honest lieutenants will get the proper command.

- Mark

on December 1st, 2008 at 4:15 pm, Paul said:

Hi Mark,

I agree. That's kind of what I meant: they agree on something, but this is what was bugging me… one last question (I hope) as a followup. Let's assume three things: (1) The commanding general is the only traitor. (2) A traitor can put out any single character, not just X and 0. (3) There are 4 generals (so N >= 3M + 1). The commanding general sends a different order value to each of the lieutenants (ie, '1′='Attack' to lieutenant #1, '2′='Stay Put' to lieutenant 2, and '3′='Retreat' to lieutenant 3), and each of these loyal lieutenants report exactly what they were told to the others… how do they have any kind of agreement? Yet this scenario seems to fit within the parameters of a succesful execution of the problem? That's why I assumed there must be some default action they take. If I am correct in my assumption (which I may not be) then the protocol is limited to binary orders/messages? I've thought of even more involved possibilities than this.

Paul

on December 1st, 2008 at 4:26 pm, Paul said:

@Mark,

I found the answer to my own question! :-) It's in the original paper:

"A traitorous commander may decide not to send any order. Since the lieutenants must obey some order, they need some default order to obey in this case. We let RETREAT be this default order."

By extension, I believe this should be extended to "In the presence of a tie in the majority voting, use the default order." This would handle the case I presented above, and others similar to it.

Thanks,

Paul

on March 31st, 2010 at 2:15 pm, Schwinn said:

The example shown with two faults is illustrative. However, as far as I can tell from the original algorithm, there is no "roll up" of the output values as described above. Notice that the algorithm does not use the term "output" at all.

Instead, the input values received in the previous step are used for the majority decision (in step 3 of the algorithm). For example, in Figure 5, I think the correct "outputs" would be the same as shown, but they would be calculated differently. As an example, the output of {0, 12, ?} in OM(1) would be calculated from the outputs of {0, 132, 0}, {0, 142, 0}, {0, 152, 0}, {X, 162, X}, and {X,172, X}, because those are the input values received by 2 in OM(0).

 on March 31st, 2010 at 2:39 pm, Mark Nelson said:

@Schwinn:

>as far as I can tell from the original algorithm,

Good luck actually taking the original algorithm description and putting it into practice! I find it to be a bit elusive.

However, I do believe my implementation is faithful to the spirit and the letter of the original paper.

- Mark

> on March 31st, 2010 at 2:56 pm, Schwinn said:

Unlike your description, the original algorithm only sends values, does not return values. However, I can't tell for sure if your description achieves the same effect.

> on May 23rd, 2010 at 9:34 pm, Byzantine Generals Problem Coding Sample at ZenSRC said:

[...] Several months ago, while going through the motions of updating my resume package, I realized that I didn't have an up to date coding sample. So began the process of becoming intimately familiar with the Byzantine Generals Problem. For a great analysis of the problem and one potential implementation, check out Mark Nelson's post on this very problem. [...]

> on December 30th, 2010 at 6:33 pm, aureliano said:

Hi Mark!

thanks for the opportunity to discuss the algorithm here. Your analysis help me a lot understanding what Lamport meant. What Swinn says, though, is in my opinion correct.

What we see in your tree as leaves in Figure 5 — OM(0) — are the messages sent already by everyone. So everyone is possesing these. Now Lamport says "let v_j be the value Lieutenant i received from Lieutenant j", and then "Lieutenant i uses the value majority (v_1, … , v_{n-1})".

Thus, the messages {0, 132, 0}, {0, 142, 0}, {0, 152, 0}, {X, 162, X}, and {X,172, X} were sent by Lieutenant 2, which itself reveived it from Lieutenant 4, 5, 6, 7! So we need to apply the mayority function on this messages, to know, what node 2 will decide.

Really interesting =)

Hope to get an answer if I might be wrong.

Best regards
aureliano

on March 4th, 2012 at 6:41 am, Quora said:

If I'd like to have an informed opinion about Bitcoin which books should I read?…

The field is moving so quickly that research articles and wikis are probably more important than books at this point. The bitcoin protocol is built using well-known cryptographic primitives so the specific advance was more related to the distributed na…


on October 24th, 2012 at 5:12 pm, AlexJF said:

Thank you very much for your explanation!

The lack of examples for N > 4 and an apparent counter-example I came up with where I was able to reach agreement with N = 6 and F = 2 with a simple majority algorithm were leaving me perplexed as to why one would need 3F + 1 nodes to reach agreement.

This part, in particular: "a subversive source process with one collaborator can cause half the processes to choose to attack, while half the processes elect to retreat, leading to maximum confusion" was mind-opening. I had forgotten to test a faulty commander with N = 6 and M = 2 and considering that case I can indeed reach a non-agreement state using the simple majority algorithm. In fact, I can also reach such a non-agreement state with N = 7 (due to even N – 2, we can have half of the N – 2 saying 0 and the other half saying 1).

From what I figure, in order to being able to get a majority for that case, Lamport's algorithm has the side-effect of not being able to obtain agreement in N = 6, M = 2 with non-faulty commander where the simple majority algorithm works.

Thank you once again for your detailed analysis and explanation :)

on February 13th, 2013 at 1:01 pm, omar000 said:

I am a bit confused about the number of iterations m+1 . why is this condition ?
and in your first example. you pick m=1 and you did 3 iterations ?
In the first round, the message from general to all soldiers. In second round each soldier sent 6 messages including himself . and then last round where you told that each will send 36 messages.
can you explain me why 3 iterations because it should have 2 iterations according to rule m+1 ?
thanks


on February 13th, 2013 at 2:29 pm, Mark Nelson said:

@omar00:

This can be a confusing algorithm, no doubt about it.

Why are there m+1 iterations? Because Lamport proved that his algorithm was correct after m+1 iterations. If there are m faulty processes and the algorithm were only to go through m iterations, it would be possible to have an incorrect result.

The proof of this is something you need to get from the original paper – this blog post is about implementation.

I only have one specific example in this article, it follow the heading "A More Complicated Example." The examples before it could be for an instance in which the maximum number of faulty processes was 2 and the the total number of processes is 7, but I didn't work that part out completely.

The example in "A More Complicated Example" is a very specific example of the message flow that occurs when n is 7 and m is 2.

- Mark

on July 3rd, 2014 at 9:59 am, Can the Bitcoin protocol morph into Virtual Collective Consciousness? | Descrier News said:

[...] a nutshell, the Byzantine General's Problem occurs in a situation where a number of agents working in a peer-to-peer relationship must coordinate their actions while [...]

on June 11th, 2017 at 9:21 am, RobertLee said:

This is _the_ most cogent and detailed explanation of the OM algorithm I've come across. Period!

on December 1st, 2017 at 9:16 am, Ouri said:

I don't get it. Why do we need two rounds in figure 4 and 5. Isn't the majority on the first round enough for all the honest lieutenants to agree on the right output?

on December 2nd, 2017 at 12:16 pm, Mark Nelson said:

@Ouri: No, there are two things that can wrong with an early decision such as you propose.

If all the generals had good information at that point, then a simple majority would work. You know that a majority of the lieutenants are honest, so this should work.

However, some number of lieutentants have bad information at that point. Either bad information from the general himself, who may be dishonest, or from other dishonest lieutenants.

There can be a large enough number of dishonest lieutenants so that they swing the early votes the wrong way. You can work this out pretty easily.

- Mark

Leave A Reply

You can insert source code in your comment without fear of too much mangling by tagging it to use the iG:Syntax Hiliter plugin. A typical usage of this plugin will look like this:

```
[c]
int main()
{
printf( "Hello, world!\n" );
return 1;
}
[/c]
```

Note that tags are enclosed in square brackets, not angle brackets. Tags currently supported by this plugin are: as (ActionScript), asp, c, cpp, csharp, css, delphi, html, java, js, mysql, perl, python, ruby, smarty, sql, vb, vbnet, xml, code (Generic).

If you post your comment and you aren't happy with the way it looks, I will do everything I can to edit it to your satisfaction.

Username (*required)

Email Address (*private)

Website (*optional)

Post My Comment

Links From Google

Popular Posts

Recent Comments

- **Ernst Berg** in The Random Compression Challenge Tu…
- **Ernst Berg** in The Random Compression Challenge Tu…
- **Ernst Berg** in The Random Compression Challenge Tu…
- **Mark Nelson** in The Random Compression Challenge Tu…
- **Ernst Berg** in The Random Compression Challenge Tu…
- **Ernst Berg** in The Random Compression Challenge Tu…
- **Ernst Berg** in The Random Compression Challenge Tu…
- **Paul** in The Random Compression Challenge Tu…
- **Ernst Berg** in The Random Compression Challenge Tu…
- **Ernst Berg** in The Random Compression Challenge Tu…

Feeds

Main Feed
Main Comment Feed
This Article's Comment Feed

Categories

Security Cisco Windows Programming VoIP Hackery Humor Standards Networking Serial Communications Video Puzzles Scams Linux Culture Graphics Uncategorized Work Writing Audio Snarkiness Mathematics Web Articles People Business Complaining C/C++ Programming Computer Science Magazine Articles Data Compression

Recent Entries

- Data Compression With Arithmetic Coding
- Highlights of ISO C++14
- C++ Generic Programming Meets OOP – std::is_base_of
- Debugging Windows Services Startup Problems
- My Big Company Code Interview
- One Definition to Rule Them All
- How I Spent My Last Few Weeks
- The Random Compression Challenge Turns Ten
- C++11: unique_ptr<T>
- C++11 – Threading Made Easy

My Books

- The Data Compression Book
- Serial Communications: A C++ Developer's Guide, 1st. ed.
- Serial Communications: A C++ Developer's Guide, 2nd ed.
- C++ Programmer's Guide to the Standard Template Library
- Developing Cisco IP Phone Services: A Cisco AVVID Solution

Archives

Select Month