CS 138: Byzantine Consensus

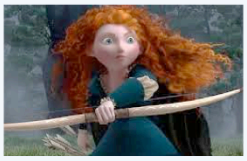
This lecture is based on "The Byzantine Generals Problem," a classic paper by L. Lamport, R. Shostak, and M. Pease. It appeared in ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3 (July 1982). Within Brown it can be found at http://delivery.acm.org/10.1145/360000/357176/p382-lamport.pdf?key1=357176&key2=2878731721&coll=ACM&dl=ACM&CFID=58767845&CFTOKEN=87710267. It is discussed in Coulouris et al. in Section 15.5.

Byzantine Generals Problem

Commanding General

Attack!

Attack!



Lieutenant General

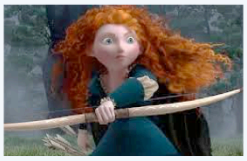
Lieutenant General

Byzantine Generals Problem

Commanding General

Retreat!

Retreat!



Lieutenant General

Lieutenant General

Byzantine Generals Problem

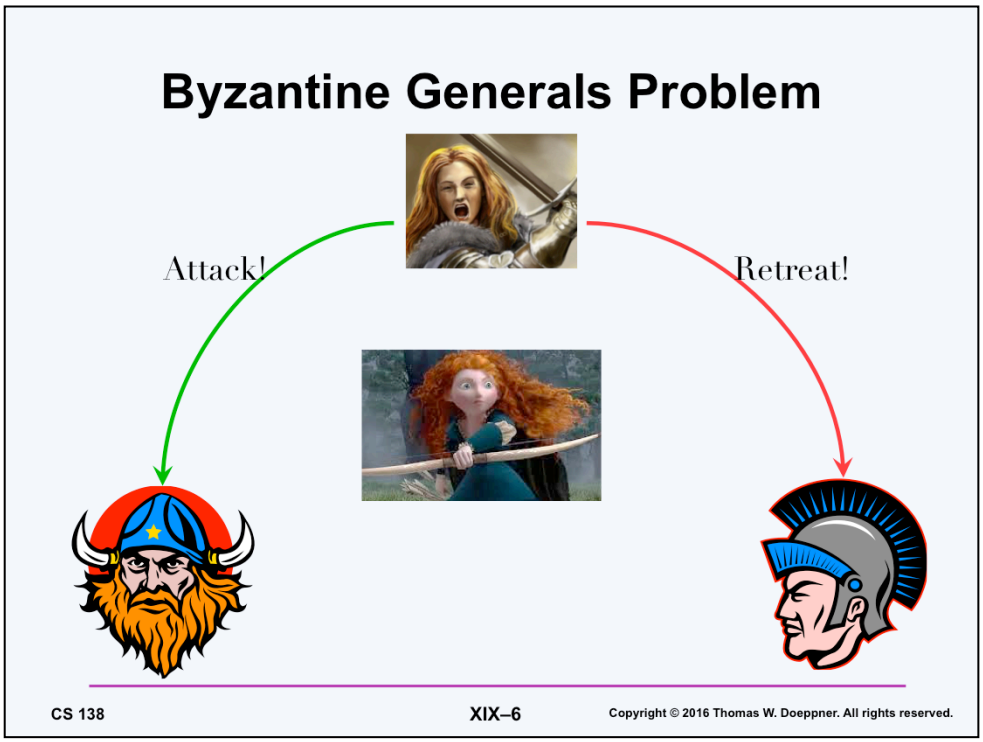
- C1: All loyal lieutenant generals obey the same order
- C2: If the commanding general is loyal, then every loyal lieutenant general obeys the order she sends

Byzantine Agreement Problem

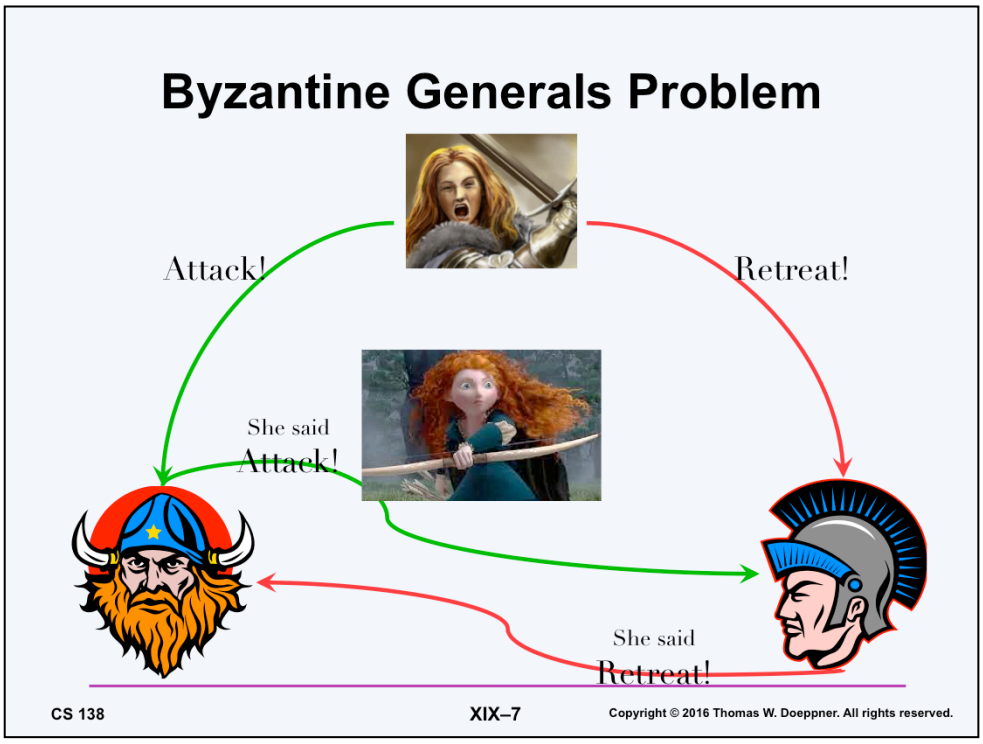
- All generals co-equal
  - each general *i* has a value *v(i)* she sends to the others

1) Every loyal general must obtain the same information *v(1), … , v(n)*
2) If the *i*th general is loyal, then the value she sends must be used by every loyal general as the value of *v(i)*

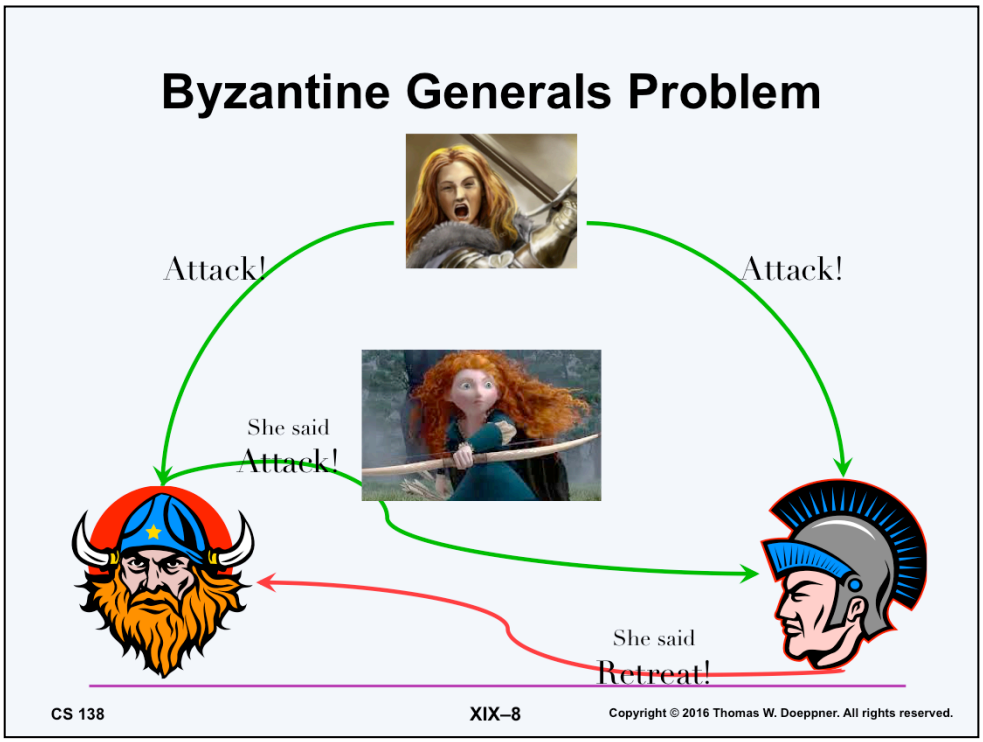
This version of the problem can be solved by doing the Byzantine Generals Problem n times concurrently, once with each general as the commander and the others as lieutenants.



Here the commander is a traitor.



To check whether a traitor is in their midst, the lieutenants exchange messages stating what they heard from the commander.



Of course, it might not be the commander who's a traitor, but one of the lieutenants. Note that, from the point of view of the bottom-left general, there's no difference between this scenario and the one of the previous slide. He can't be sure, if he attacks, that there will be another general joining him. Though this isn't a formal proof, it should be convincing evidence that with three generals of whom at most one is a traitor, the Byzantine generals problem has no solution.

Summing Up

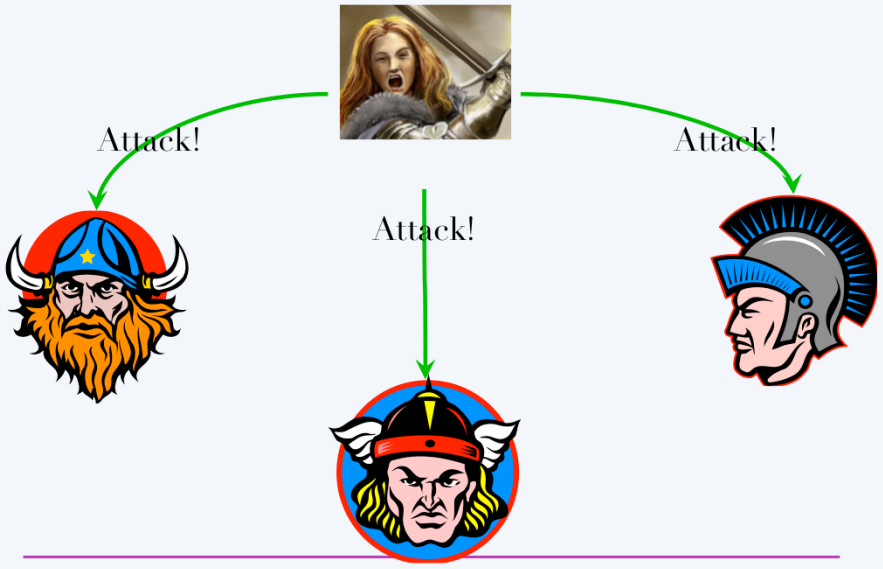
- Byzantine Generals Problem with 3 Generals, at most one of whom is a traitor ([3,1]BGP)
  - no solution satisfying C1 and C2

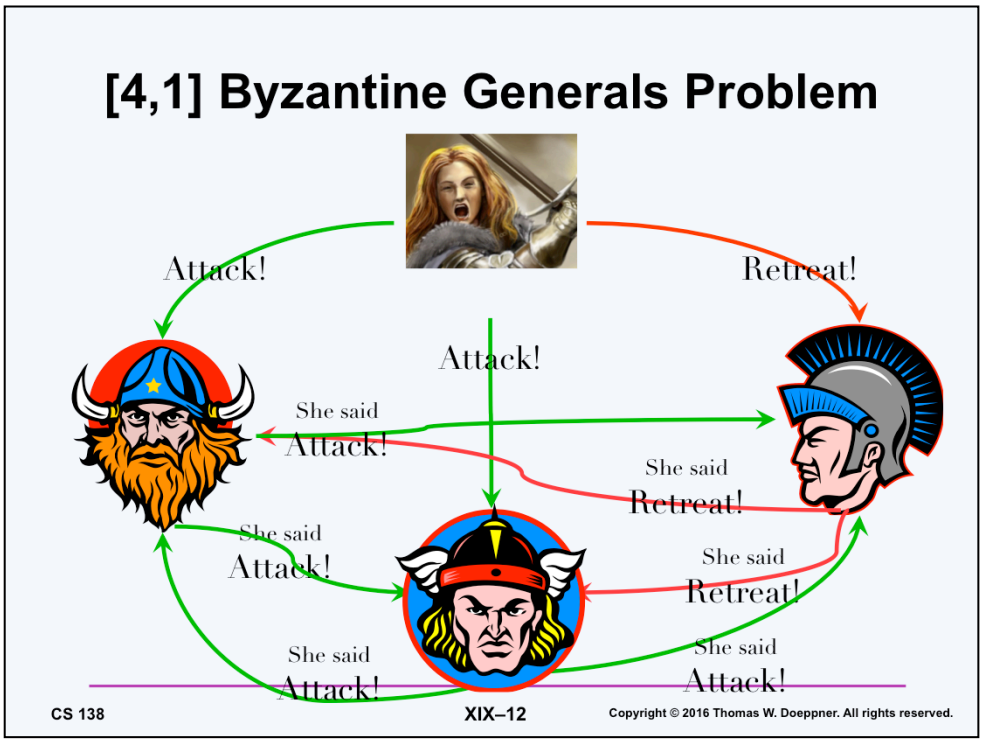
Message-Passing Assumptions

- Every message sent is delivered correctly
- The receiver of a message knows who sent it
- The absence of a message can be detected

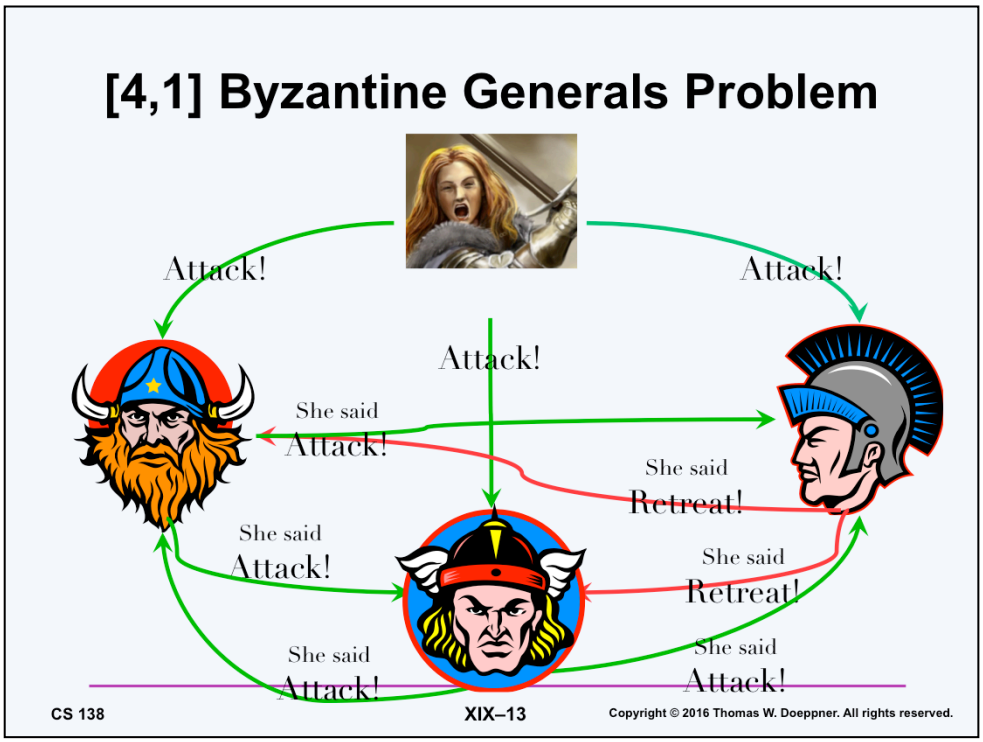
The last assumption is satisfied in *synchronous* systems.

[4, 0] Byzantine Generals Problem





Here we have four generals, with at most one of them a traitor. In this case, the traitor is the commanding general.



Again we have four generals, but the traitor is the rightmost general. The two lieutenant generals can't distinguish this situation from the one of the previous slide. However, in both cases, since the majority of messages received say attack, they can feel confident that if they attack, they will be joined by two other generals.

Some Details

- Each general receives messages u, v, and w from the others
  - if no message is received, interpret its lack as "retreat"
- Loyal general takes its order to be majority(u, v, w)
  - if no majority: retreat

Summing Up

- Byzantine Generals Problem with 4 Generals, one of whom is a traitor ([4,1]BGP)
  - solvable

Theorem

- If N is the number of generals and T is the number of traitors, then there is a solution to the Byzantine Generals Problem iff

$$N > 3T$$

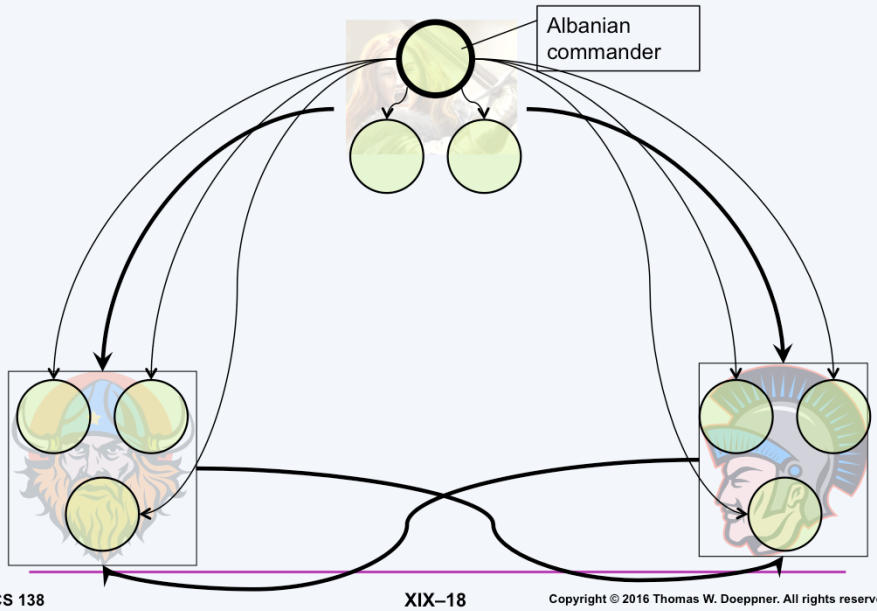
The assumptions here are that communication is synchronous and that messages are unsigned. We'll modify these assumptions soon.

Proof

- Only if:
  - assume a solution exists for N ≤ 3T
    - 3T Albanian generals can cope with T traitors
  - three Byzantine generals now take advantage of the Albanian approach to solve [3,1]BGP
    - commander simulates Albanian commander plus at most T-1 lieutenant generals
    - two lieutenant generals each simulate at most T Albanian lieutenant generals
  - loyal Byzantine generals simulate loyal Albanians
  - traitorous Byzantine general does whatever it takes to mess things up
    - effectively simulates actions of up to T traitorous Albanian lieutenant generals

The reference to Albanians comes from the original paper …

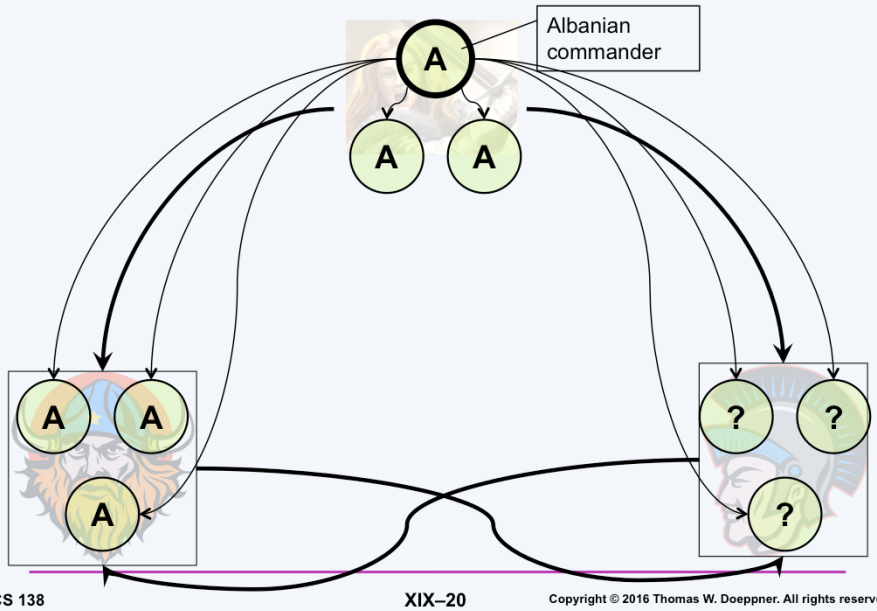
Albanian Simulation



Proof (Continued)

- By C1: all loyal Albanian lieutenant generals obey same order
  - thus loyal Byzantine lieutenant generals obey orders obeyed by simulated Albanians
- By C2: if Albanian commander is loyal, then all loyal Albanian lieutenant generals obey her order
  - thus if Byzantine commander is loyal, her order is that of Albanian commander

Albanian Simulation



Proof (Half Done)

- This gives us a method to solve [3,1]BGP
  - which can't be done …

Proof (remainder)

- If:
  - Show that a solution exists if N > 3T
    - T=1
      - done
    - T>1
      - hard
        - next few slides

[7,2]BGP

- Case 1: the commander is loyal
  - six lieutenants receive order *v*
  - four report it to one another correctly
  - two (traitors) do not
  - correct outcome determined by majority
  - (that was easy!)

[7,2]BGP (continued)

- Case 2: the commander is a traitor (and so is someone else)
  - not so easy …
  - if the commander is a traitor, there is only one traitor among the lieutenants, so they can work out agreement assuming only one traitor
    - this is the Byzantine agreement problem, which means each lieutenant runs the algorithm

The Algorithm, part 1

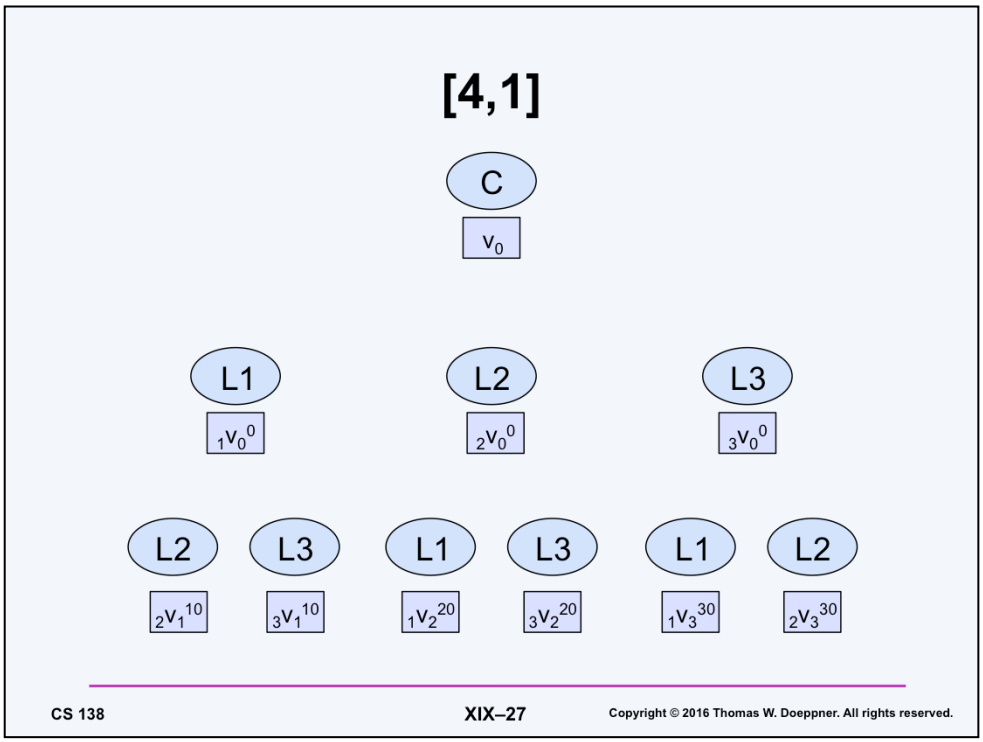
- BGP(0) // no traitors
  1) the commander sends her value to each lieutenant
  2) each lieutenant uses the value he receives from the commander

This is from the aforementioned paper by Lamport, Shostak, and Pease. It's easy to analyze, but it's not a well formed algorithm …

The Algorithm, part 2

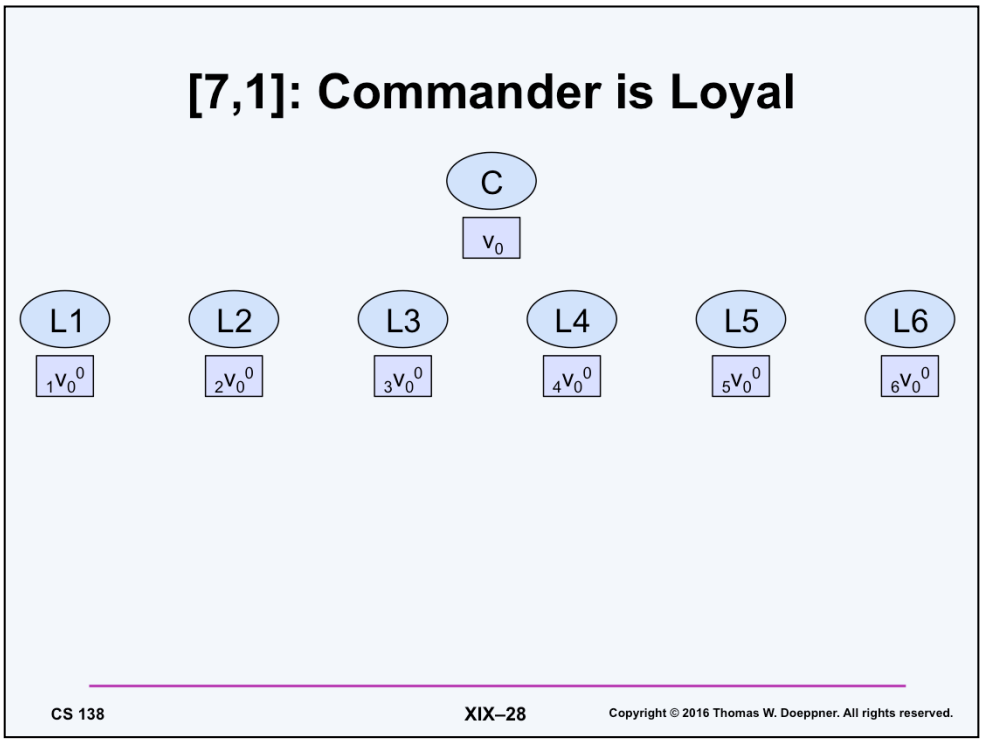
- BGP(m) // m traitors
  1) the commander sends her value to each lieutenant
  2) for each *i*, let v_i be the value lieutenant *i* receives from the commander. Lieutenant *i* acts as the commander in BGP(m-1) to send v_i to each of the *n-2* other lieutenants
  3) for each *i* and each $k \neq i$, let v_k be the value lieutenant i received from lieutenant *k* in step 2 (using BGP(m-1)). Lieutenant *i* uses the value *majority*(v_1, …, v_{n-1})

This is from the aforementioned paper by Lamport, Shostak, and Pease.

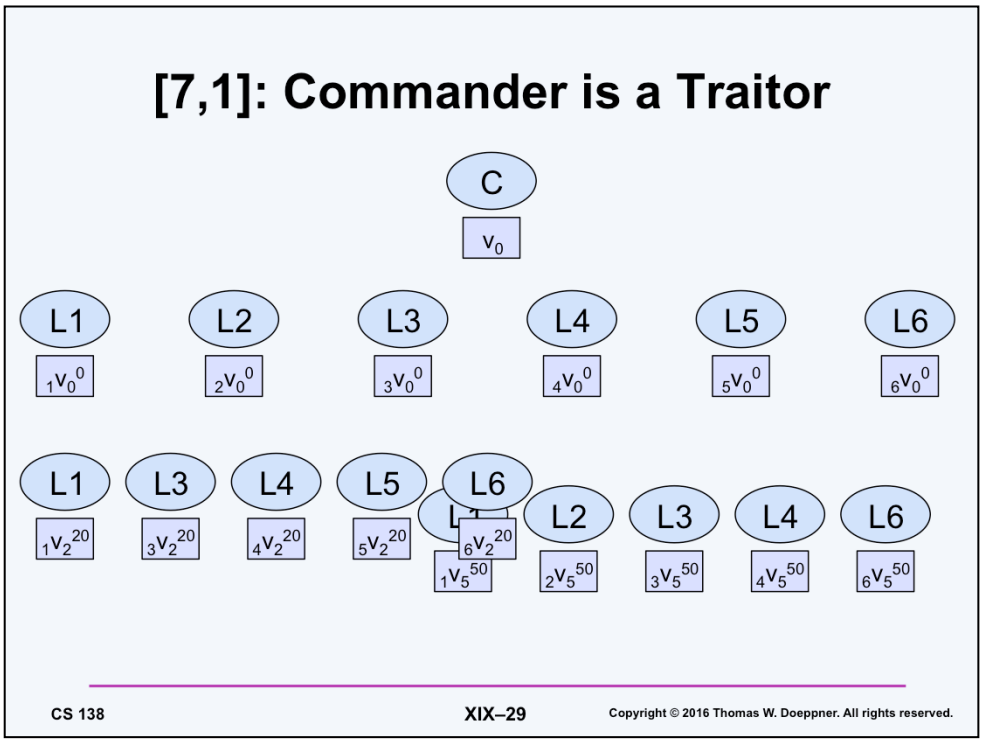


In this slide we work out the notation we'll be using for [7,2] (and beyond).

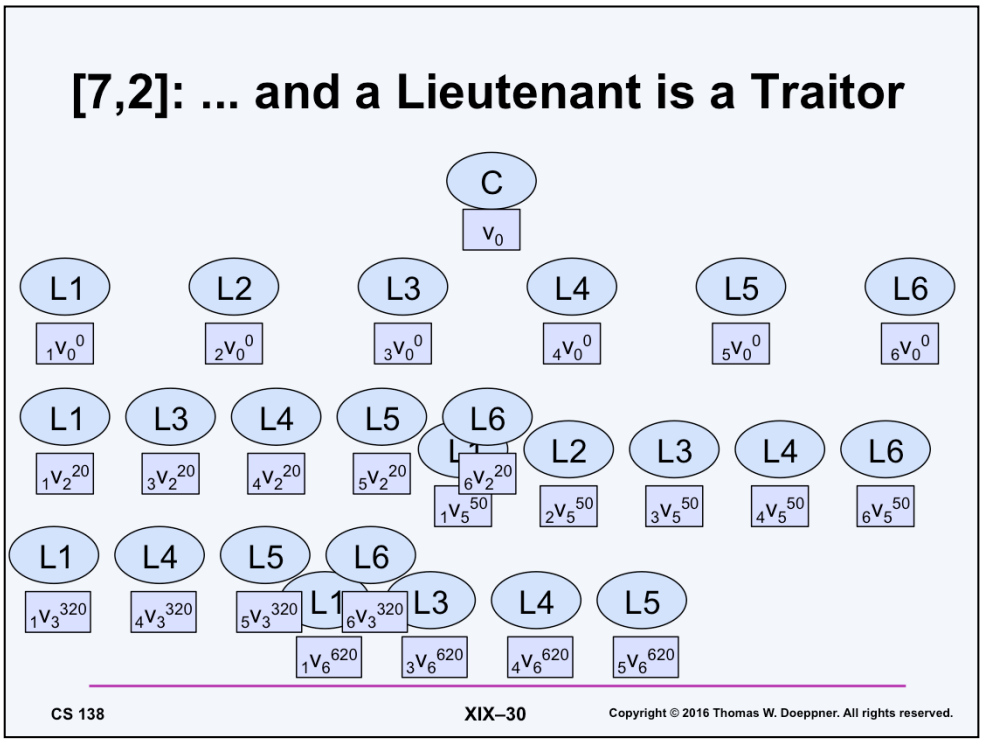
The commander, C, sends its order (v_0) to the lieutenants. Lieutenant i stores it in $_iv_0^0$. The Lieutenants then send their values to the others. The value Lieutenant k receives from Lieutenant m is placed in $_kv_m^{m0}$. The intent is that the superscripts show the paths through the tree. Lieutenant 1 follows the order *majority*($_1v_0^0$, $_1v_2^{20}$, $_1v_3^{30}$); lieutenant 2 follows the order *majority*($_2v_0^0$, $_2v_1^{10}$, $_2v_3^{30}$); lieutenant 3 follows the order *majority*($_3v_0^0$, $_3v_1^{10}$, $_3v_2^{20}$).



Consider now BGP with 7 generals, two of whom might be traitors. If we're assured the commander is not a traitor, then each lieutenant can simply use the order received from the commander.



If the commander could be a traitor, then the lieutenants must check with one another what order was actually sent. The slide shows the result of lieutenants 2 and 5 communicating with the others. If none of the lieutenants are traitors, we need go no further (though all the other lieutenants must communicate the orders they received as well). But, of course, it's not known whether any of the lieutenants are traitors.



But if both the commander and one of the lieutenants may be traitors, then more work is required. Here lieutenant 2 has communicated the order it received from the commander to the others, as in the previous slide. However, since one of the lieutenants may be a traitor, no one trusts lieutenant 2 to have communicated the same order to each of the others. We now have another instance of BGP, this time with at most one traitor (since we're already assuming the commander is a traitor). Thus the other lieutenants communicate with one another the order they received from lieutenant 2 (effectively saying to one another "lieutenant 2 said that the commander said …"). Of course we have to do this for each of the lieutenants, so the complete diagram gets rather large.

A Better (?) Algorithm …

```
BGP(m, gens, v, path, sender) {
   if (m > 0) {
         // tell others what was received
         for each g in gens−me
               sendmsg(g, BGP, m−1, gens−me, v, me·path, me)
         // wait till all resulting communication is complete
         when defined((∀g∈gens) meV_g^(g·path) )
               // compute consensus value
               meV_sender^path = majority(v, (∀g∈gens) meV_g^(g·path) )
   } else {   // m == 0
         meV_sender^path = v
   }
}
```


Here's an attempt at a correct algorithm for BGP. It's initially invoked by the commander. m is the maximum number of traitors. gens is the set of generals (initially including the commander). v is the order (value). path is the path taken from the root (i.e., from the commander), initially empty. sender is the ID of the invoker, where 0 is the commander.

The *sendmsg* routine sends a message to the general given by the first argument. That general is to execute the command (i.e., procedure) given in the second argument (BGP), with the following arguments.

Note that the recursion proceeds in a breadth-first rather than a depth-first manner. After a sequence of calls to sendmsg, the caller does not wait for the results of these invocations to complete, but for the results of the invocations of sibling subtrees. In other words, the invocations a general makes of BGP are to send messages to other generals. What the general waits for is for the other generals to send messages to it. This waiting is handled by the "when defined" statement, which waits for the set of variables given as arguments to be given values. In principle it could be implemented as a sequence of operations on a set of semaphores, one for each variable.

Complexity

- How expensive is the algorithm for BGP?
  - T+1 rounds of messages
  - $O(N^T)$ messages, for N generals and T traitors
- Can we do better?
  - T+1 rounds are required
  - polynomial algorithm exists, but for N > 4T
    - next few slides …

An Even Better Algorithm

- Agreement on one of two values
- T traitors; T+1 phases; N > 4T
- In each phase, a different general is the commander
  - all generals broadcast values to one another
  - recipients determine "majority"
    - commander's value is tie-breaker
- In at least one phase, the commander is loyal
  - consensus reached in this phase
  - doesn't change in subsequent phases

This algorithm is from "Cloture Votes: n/4-resilient Distributed Consensus in (T+1) Rounds," P. Berman and J. Garay, Mathematical Systems Theory, 26(1), 1993.

Details

```
for (phase = 1; phase <= T+1; phase++) {
    // round 1: executed by each general
    broadcast value to all others
    await value v_j from each general G_j
    majority = value that occurs > N/2 times
          default value otherwise
    mult = number of times majority occurs
```


This presentation is taken from "Distributed Computing: Principles, Algorithms, and Systems," by Ajay D. Kshemkalyani and Mukesh Singhal, Cambridge University Press, 2008.

Details (2)

```
    // round 2: executed by each general
    if (this is G_phase)
        // G_phase is (temporary) commander
        broadcast majority to all other generals
    else
        receive tiebreaker from G_phase
        if (mult > N/2 + T)
                value = majority        // super majority
        else
                value = tiebreaker
}
```

Correctness

- Assume commander in phase p is loyal
  - its value x (from round 1) is either majority or default value
  - it broadcasts x in round 2
- Claim 1: all loyal generals (including phase p commander) agree on value
  - proof: soon
- Claim 2: if all loyal generals agree on value at beginning of phase i, they agree at end of phase i
  - proof: soon
- After phase T+1, all loyal generals agree

Claim 1

- All loyal generals (including phase p commander) agree on value
  - consider all pairs of loyal lieutenants G_i and G_k
  - they can set their values in one of three ways:
    - both set their value to the (super) majority
      - super majority must involve more than n/2 loyal lieutenants
      - any two such majorities must have a member in common
        - thus G_i and G_k have same value
      - G_p must have heard from same majority
        - it also has same value

Claim 1 (continued)

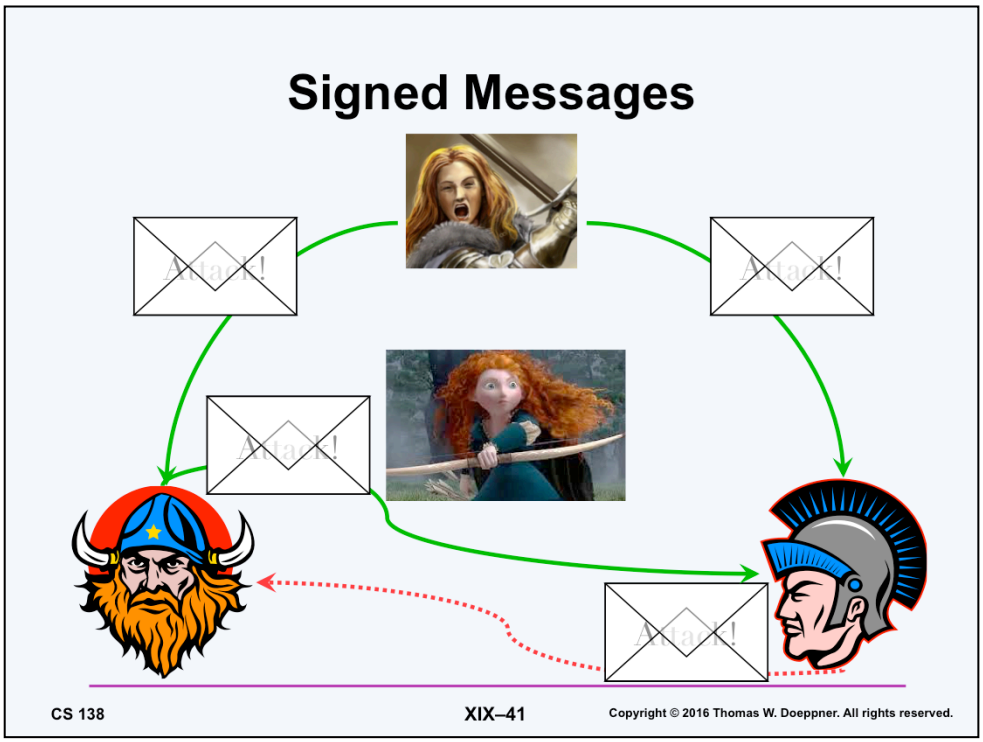
- both set their value to the commander's tie-breaking rule
  - since commander is loyal, both now agree with commander
- G_i sets value to (super) majority; G_k to tie-breaking rule
  - since super majority agrees with G_i, more than n/2 loyal nodes agree, thus G_p agrees
  - G_p value is adopted by G_k
    - i.e., this case is same as first case

Claim 2

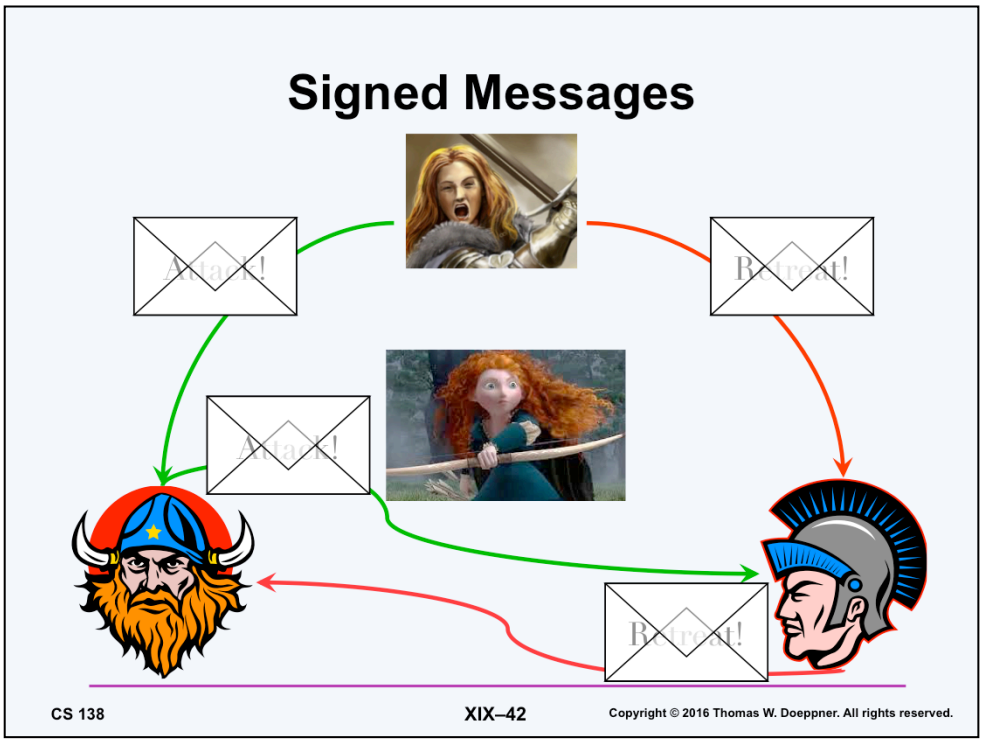
- If all loyal generals agree on value at beginning of phase i, they agree at end of phase i
  - all generals receive consensus value from a majority of others in round 1
  - thus all loyal generals stay with this value in round 2

Complexity

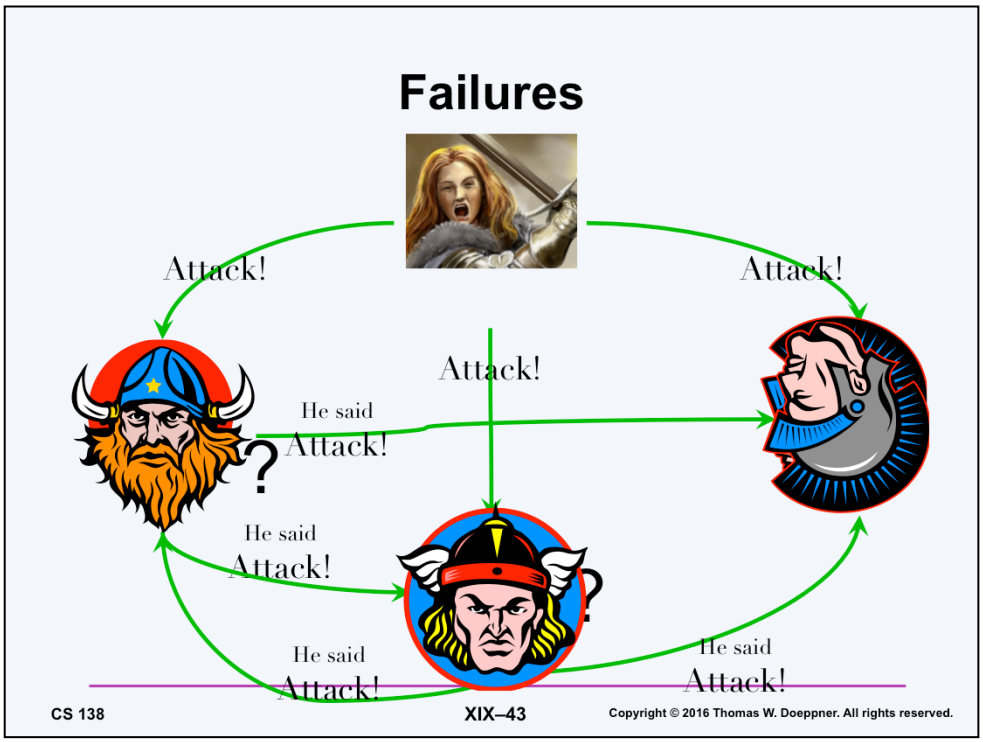
- T+1 phases
- n·(n-1) messages in round 1
- n-1 messages in round 2



Here the commander cryptographically signs her messages and the lieutenants are required to send copies of the signed messages to the others. Thus a traitorous lieutenant cannot lie about what the commander sent.



Here both lieutenant generals realize the commander is a traitor. However, the two of them must nevertheless come to a consensus. If there isn't a clear majority for attack or retreat (as is the case here), then the convention is that they retreat. Thus they have a consensus.



Here one lieutenant has failed to respond. In a synchronous system, we can use a timeout to decide that a process has failed and thus, in this case, interpret the silence as a “retreat”.

Asynchronous Communication

- Processes may respond to messages at arbitrary times
  - can't use timeouts to determine failures
- BGP has no solution
  - non-responding general might respond at any time with whatever response counters the decision made assuming it was missing
  - in practice this is surmountable

Surmounting Failure

- Recover quickly
  - state kept in non-volatile memory
- Detect failure
  - enforced timeouts
- Be unpredictable
  - randomized algorithm