Math 161 - Verifying the Byzantine Generals Problem

Mark Pekala

April 30, 2023

Contents

1 Mathematical Background

The results discussed in this paper were originally presented by Leslie Lamport and others in 1980 [1]. The problem arises when thinking about how groups of computers in large systems may (or may not) work together to achieve complex tasks. In this section, we start by developing intuition for the problem through an informal analogy and a simple example. Then, we present a formal statement of the theorem and it's proof to place the problem on solid mathematical footing.

1.1 Informal Statement

The name "Byzantine Generals" comes from the following analogy that is often used to explain the problem at a high level. Imagine you have a group of three Byzantine generals who are planning an attack on a city. To complete the siege, each must position themselves on a different end of the city, so that the only way to communicate between them is via messengers. We assume that messengers travel around the city on horseback and are reasonably fast, and that each general may send as many messages as they like to the other generals containing arbitrary amounts of information. It is also assumed that generals recognize messengers, and thus know who sent them information.

In order for the attack to be successful, the generals must agree on a precise time to attack the city *after* they have already split up into three groups. They cannot agree on a fixed time beforehand because the time may depend on what they observe about defenses at their various posts. To make matters worse, one of the generals is secretly a traitor who is trying to send incorrect messages to sabotage the attack.

If the two non-traitorous generals can agree on who the traitor is and a time to attack, they will succeed in overwhelming the defenses. If they cannot agree, they will attack at different times and all will be lost. The problem is then as follows: **is there a way for the non-traitorous generals to communicate using only messages that guarantees success?**

This problem is important in the field of computer science because it helps us reason about how much duplication is necessary to prevent bad effects from certain kinds of faults. The traitorous general models a machine exhibiting undefined behavior as a result of things like hacking, broken hardware, etc.

There is a rather simple, but informal, proof of impossibility in the example described above. Assume for the sake of contradiction that reaching agreement is possible, and take the perspective of one of the non-traitorous generals. There must exist a sequence of messages that the other non-traitorous general can send to both agree on a value, and convince us that the other general is a traitor. However, then the other (traitorous) general would be able to use a similar but slightly different series of messages to convince us that the other general was faulty, and give us a different time to attack. With only three generals, there is nothing we can do to distinguish between the two, and thus such a successful protocol does not exist and attacking is futile.

The problem can be extended to more generals, and can be phrased by assuming that we have a group of n generals, f of which may be traitors, and asking whether a sequence of communications can guarantee success in this case. For the purpose of this paper, we stick to the case when $n = 3$ with one traitor, and precisely formalize this impossibility proof.

1.2 Definitions

Next, we aim to leave the world of 15th century combat and provide concrete definitions that can adequately describe the situation presented above. Note that while some of these definitions are similar to those presented by Lamport[1], many are new or use different objects to maximize compatibility with Lean.

Definition 1. A *processor* is simply a natural number $n \in \mathbb{N}$. A groups of *processors* is a finite set $P \subset \mathbb{N}$.

Definition 2. Given a group of processors P, we define a *truth map* $t : P \to \{0, 1\}$ to be a representation of which processors are faulty. For all $p \in P$, we take $t(p) = 1$ to mean that the processor is truthy, and $t(p) = 0$ to mean that the processor is faulty. Note that the *truth map* exists for the system, but is not known by any of the processors individually.

Definition 3. The *value map* is a function $v : P \to \mathbb{N}$ that represents the private information of each processor. Again, v exists for the system as a whole, but for each processor p, only the value $v(p)$ is known before communication.

Definition 4. A *communication* is a finite sequence (list) of elements in P representing the order of information flow. This is meant to generalize so that if $P = \{1, 2, 3\}$, then the sequence $[1, 3, 2]$ would represent 1 sending a message to 3, and then 3 sending the information it received from 1 to 2. Numbers are allowed to repeat, so that $[1, 2, 3, 2, 3]$ is also a valid communication. We let $\mathcal{L}(P)$ represent the set of all finite length sequences of elements of P (i.e., all possible communications), and $\mathcal{K}(P)$ represent all finite length sequences of elements of $\mathcal{L}(P)$. For example, an element of $\mathcal{K}(P)$ might look like

$$[[1, 2, 3], [1, 3], [1, 2, 3]]$$

which helps us represent the fact that communications happen in order and that the same communication can happen multiple times in the same protocol.

Definition 5. A *protocol* is a combination of a *communication record* $W : \mathcal{K}(P)$ and a *network function* $\sigma : \mathcal{L}(P) \to \mathbb{N}$, such that if a list $[q_1, q_2, \ldots, q_k]$ satisfies $t(q_i) = 1$ for all i, then $\sigma([q_1, \ldots, q_k]) = v(q_1)$.

The communication record W represents all communications that happen in the system. For example, $W = [[2, 1], [3, 1], [2, 3, 1]]$ represents a scheme in which 2 communicates it's value directly to 1, 3 communicates it's value directly to 1, and then 2 communicates it's value through 3 to 1.

The network function σ represents the fact that truthy processors tell the truth. That is, when communications contain all truthy processors, whoever is at the end of the communication will successfully learn the private value of the first processor in the communication.

Definition 6. An algorithm is a function $A : (\mathcal{K}(P), \mathcal{L}(P) \to \mathbb{N}) \to \mathbb{N}$. Logically, it takes in a list of communications, a network function, and outputs a guess at one of the values in the value map.

Definition 7. An algorithm is *sound* if $A([], \sigma) = 0$ for all σ.

Definition 8. An algorithm A is *blind* if for all $K \in \mathcal{K}(P)$, and for all valid network functions σ and σ', the existence of a function $m : \mathbb{N} \to \mathbb{N}$ satisfying $\sigma'(l) = m(\sigma(l))$ for all $l \in K$ implies that

$$A(K, \sigma') = m(A(K, \sigma)).$$

This notion of blindness was not present in the original paper, but is crucial for the formalization of the proof. In essence, it requires that the algorithm depend strongly on the values it receives, so that if it receives the same kinds of communications but containing different numbers, it will reach predictably different outcomes.

Definition 9. The function StartsWith : $(P, \mathcal{K}(P)) \to \mathcal{K}(P)$ takes a processor q and a list of lists of processors, and filters that list to contain only the lists of processors that start with q.

Definition 10. The function EndsWith : $(P, \mathcal{K}(P)) \to \mathcal{K}(P)$ takes a processor q and a list of lists of processors, and filters that list to contain only the lists of processors that end with q.

Definition 11. The function FilterBy : $(P, P, \mathcal{K}(P)) \to \mathcal{K}(P)$ takes two processors p and q and a list of lists of processors, and filters that list to contain only the lists of processors that start with p and end with q.

Definition 12. A pair of a protocol $T = (W, \sigma)$ and algorithm A is said to be *interactive* with respect to the value function v and truth map t if

$$\forall p, q \in P, t(p) = t(q) = 1 \implies A(\text{FilterBy}(q, p, W), \sigma) = v(q).$$

Simply put, a protocol algorithm pair is interactive if truthy processors correctly compute the private value of other truthy processors.

Definition 13. A pair of a protocol $T = (W, \sigma)$ and algorithm A is said to be *consistent* with respect to the value function v and truth map t if

$$\forall p, q \in P, t(p) = t(q) = 1 \implies \forall x \in P, A(\text{FilterBy}(x, p, W), \sigma) = A(\text{FilterBy}(x, p, W), \sigma)).$$

In other words, the pair is consistent if every truthy processor computes the same guess at the value map. Note that this requires that the truthy processors must agree on the value they assign to faulty processors as well, which in practice usually means that all truthy processors correctly find out the faulty ones.

Armed with these definitions, it's now possible to unambiguosly state the main result.

Theorem 1. ***Byzantine Generals (n=3).*** *Let $P = \{1, 2, 3\}$. For all communication records W and algorithms A that are sound and blind, there exists a truth map t, value map v, and networking function σ such that $t(p) = 0$ for exactly one $p \in P$, σ is a valid networking protocol, and the pair $((W, \sigma), A)$ is not interactive or not consistent.*

Formally proving the above result on $n = 3$ is the main goal of this paper. It formally states what we mean when we say there is no algorithm allowing three generals to agree in the face of one fault. Below, we state the general result for arbitrary n, which implies that the problem is solvable when $n \geq 3f + 1$ where f is the number of faults. We do not address this general case in this paper.

Theorem 2. ***Byzantine Generals (arbitrary n).*** *For all groups of processors P, there exists a communication record W and an algorithm A that is sound and blind such that for all truth maps $t : P \to \{0, 1\}$ with $|\{p : P \mid t(p) = 1\}| \geq 2$, $|P| \geq 3 \cdot |\{p : P \mid t(p) = 0\}| + 1$ **if and only if** for all value maps v and network functions σ satisfying network function properties, the pair $((W, \sigma), A)$ is interactive and consistent.*

1.3 Proof of Impossibility in the Case $n = 3$

In this section, we present a formal proof of the Byzantine Generals theorem for $P = \{1, 2, 3\}$.

Let W be a communication record and A be an algorithm.

We consider two cases.

Case 1: $\exists p, q \in P$, such that $p \neq q$ and $\text{FilterBy}(q, p, W) = []$. In other words, this is the case when the communication record includes two processors p and q such that q never even attempts to send it's value to p (directly or indirectly).

In this case let x be the third element in P besides p and q. We pick t such that x is the one faulty processor. Our value map will simply be the function $f(r) = 1$ for all processors r. Our networking function σ will send the empty list to 0, and all other lists to 1. Clearly σ satisfies the property of a network function, since it maps any list of all truthy processors to 1, which is correct for the value map.

Notice that $\text{FilterBy}(q, p, W)$ is empty. By the soundness of A we thus must have

$$A(\text{FilterBy}(q, p, W), \sigma) = 0 \neq v(q) = 1,$$

breaking interactivity.

Case 2: In this case, we are guaranteed that every processor talks to every other processor at least once.

Assume for the sake of contradiction that there exist a W and A which can lead to both interactivity and consistency. Use the truth map t that has 3 as the only faulty processor, and the value map $v(p) = p$. Pick σ to be any valid networking function (one can be constructed trivially).

By consistency, we know that 1 and 2 agree on the value of 3, meaning

$$A(\text{FilterBy}(3, 1, W), \sigma) = A(\text{FilterBy}(3, 2, W), \sigma) = c$$

for some $c \in \mathbb{N}$. Consider the function σ' given by

$$\sigma'([x_1, \ldots, x_k] = \begin{cases} \sigma([x_1, \ldots, x_k]) + 1, \text{ if } x_1 = 3, x_k = 1, \\ \sigma([x_1, \ldots, x_k]) + 2, \text{ if } x_1 = 3, x_k = 2, \\ \sigma([x_1, \ldots, x_k]), \text{ otherwise} \end{cases}.$$

Since $\sigma'(l) = \sigma(l)$ for all l that contain only truthy processors, σ' is still a valid network function. Furthermore, for all $l \in \text{FilterBy}(3, 1, W)$ we know that $\sigma'(l) = \sigma(l) + 1$, and for all $l \in \text{FilterBy}(3, 2, W)$ we have $\sigma'(l) = \sigma(l) + 2$.

Hence by the blindness of A we know that

$$A(\text{FilterBy}(3, 1, W), \sigma') = A(\text{FilterBy}(3, 1, W), \sigma) + 1 = c + 1$$

and

$$A(\text{FilterBy}(3, 2, W), \sigma') = A(\text{FilterBy}(3, 2, W), \sigma') + 2 = c + 2.$$

Remember that σ' is still a valid networking protocol. Thus, by assumption of correctness on W and A, we must have

$$A(\text{FilterBy}(3, 1, W), \sigma') = A(\text{FilterBy}(3, 2, W), \sigma')$$

implying $c + 1 = c + 2$, which is a contradiction.

2 The Lean Theorem Prover

2.1 Brief History

Lean is a functional programming language launched in 2013 that is built around the notion of dependent type theory. It is one of the newest tools in a long history of automated proof assistants including Isabelle, Coq, and others.

Formal verification via proof assistant is an attractive goal for a variety of reasons. One of the biggest goals of the automated proof movement is to make it possible to objectively assess the correctness of a proof without the lengthy (and sometimes sub-par) human review process. Many people also hope that one day such tools will improve the experience of writing proofs in general, as one can iterate step by step and know immediately where their argument is unsound.

2.2 Dependent Type Theory

As mentioned earlier, Lean is based around the notion of dependent type theory.

To get a quick intuition about dependent type theory and how it may be useful in automated theorem provers, we'll start with a simple example. Say that we have some universe of types $\mathcal{U}$, where one can think of elements of $\mathcal{U}$ as sets. We can simply have an element of that type $S \in \mathcal{U}$, but we can also have a function $F : S \to \mathcal{U}$ which given an element $s : S$, returns another type.

This notion is useful because of the Curry Howard Correspondence which provides intuition for how to translate formal logic into computational calculi. Let $S \in \mathcal{U}$, and $s \in S$. For any function $F : S \to \mathcal{U}$, we can think of the type $F(s)$ as a proposition. It's based around the notion that given an $s \in S$, the type $F(s)$ is inhabited if and only if s satisfies the original predicate on S.

As an example, consider the predicate m is even, for $m \in \mathbb{N}$. We can logically state this as

$$m \text{ even} \iff \exists k \in \mathbb{N} s.t. 2k = n.$$

The type corresponding to this implication is the type that takes in $m : \mathbb{N}$, and then includes another number $k \in \mathbb{N}$ satisfying $2k = n$. In type theory this is second type can be written as

$$\sum_{k:\mathbb{N}} 2k = n$$

which is dependent on the specific n chosen. Evaluating whether or not this proposition is true for a given n is then equivalent to whether or not this dependent type is inhabited. Notice as well that the question of whether the type is inhabited can be reduced to a computational question which can implemented as code and hence be solved via a machine.

Building a proof assisstant around this notion involves essentially an automated way of turning propositional logic into this computational equivalence. Once this equivalence is established, to determine if a proposition is true, one can instruct the program how to create a value of that type, which acts as a proof of the original statement.

3 Tackling Byzantine Generals In Lean

The full code used in this project can be found in the appenedix, as well as online in a dedicated git repository[2].

3.1 Formalizing "Protocol"

Part of the reason why I wanted to tackle the Byzantine generals problem to begin with is that while it's easy to understand what it says at a high level, it's not obvious how to structure it in a formal way. This is true about many important algorithms and impossibility results in computer science.

The hardest part of this specific formalization was ironing out what it meant to have a protocol for a group of n processors. Understanding things like "the set of rational numbers" or even "the set of all bijections between $\mathbb{R}$ and itself" is relatively straightforward. But what does "the set of all possible protocols between n machines connected over a network" mean? Having a precise definition was necessary as the result I wanted to prove argued that there was no such protocol that existed satisfying interactivity and consistency.

Deciding that "a group of processors" should mean "a finite set of natural numbers" was simple. Next, I needed a way of defining what communication between two machines meant. For this, I chose to use a list of processors, since it clearly defined both who participated in the communication, and the direction that information flowed. Lists also admit repitions, which capture the most general case of protocols which might involve multiple rounds of information exchange between a and b at various points in the sequence.

However, a simple list of processors is not enough. In the most generic protocol, an arbitrary number of rounds of communication may occur, so that one sequence of information is exhanged, followed by another, and another, until termination. This led me to my final definition of *communication record*, as a list of lists of information exchange.

Coming up with the *network function* σ was the next piece of the puzzle, and the first aspect of my formalization that placed restrictions on what a protocol could be. There were two things that the network function was required to capture about the setup described in the paper: (1) That for communications involving all truthy processors, a correct reading of one of the machines private values should be possible and (2) any communication involving a faulty processor can return any value, or nothing.

The reason this formalization was challenging was that at the time σ is defined, it does not have access to the actual list of truthy processors. Thus, it can't involve any calculation that knows the truthiness of a procesor. Instead, I decided to let σ be any function form a list of lists of processors to $\mathbb{N}$, but that it must carry around an additional hypothesis which takes in an arbitrary truthy assignment and verfies that communications involving all truthy processors return the first processors value, and that all other communications can return an arbitrary value.

3.2 Formalizing "Algorithm"

Coming up with a way of defining the set of all possible communication schemes involving processors was not enough. Formalizing the proof also required describing the space of all possible algorithms that each individual node may run after receiving their communications to get an arbitrary value

map.

At first, I tried to define an algorithm for a given processor as simply a function from a list of lists combined with a network function to a natural number, but this didn't work. The issue is that this definition alone does not exclude the possibility of a type of "oracle algoritm" which simply reads the values of the value map directly and returns the correct number.

To get around the fact that any arbitrary function would always include this "oracle" function that breaks the proof, I had to place restrictions on what a valid algorithm could be. First, I introduced the notion of soundness, which states that an algorithm must assign zero to an empty sequence of communication. This prevents a trivial algorithm plus protocol combination where no communication occurs and each node simply spits out the value map.

Next, I had to add in a hypothesis about how the algorithm functions. In the original informal proof, there is a notion that one may trick a truthy processor by running the same sequence of communications, but swapping the values. To formalize this, I introduced the hypothesis that the algorithm must be blind. That is, if it receives a list of communications, and a network function, and then receives that same list of communications and a new network function that is an injection on the original network function, it must return that same injection on the value. This definition allowed me to still consider each algorithm simply a function from lists of lists to numbers, without allowing algorithms that are oracles that ignore their inputs.

3.3 A Verified Proof When $n = 3$

This section highlights important aspects of the proof and their implementation in lean to show how this formal argument can be translated to an automated theorem prover.

```
1  def is_interactive {fin : finset ℕ}
2  (wrong : fin)
3  (value_map : fin → ℕ)
4  (assignment : fin → fin → ℕ)
5  := ∀ p q : fin, p ≠ wrong ∧ q ≠ wrong → assignment p q = value_map q
6
7  def is_consistent {fin : finset ℕ}
8  (wrong : fin)
9  (value_map : fin → ℕ)
10 (assignment : fin → fin → ℕ)
11 := ∀ p q : fin, p ≠ wrong ∧ q ≠ wrong → ∀ x : fin, assignment p x = assignment
       ↪ q x
```

Figure 1: Definitions for interactivity and consistency.

Next, we'll highlight the formal proof of the case when there exist two machines in the system that don't communicate.

The most important parts of this formal proof are:

- The by_cases statement instructs lean that we would like to generate two branches based on whether a proposition is true. This allows us to introduce the assumption as a hypothesis and reason from it.

```
1  -- Handle the degenerate case where some machines don't talk to each other
2  by_cases W_full : (∃ p : procs, ∃ q : procs, p ≠ q ∧ filter_by q p W = []),
3  {
4    rcases W_full with ⟨ p, q, ⟨ p_ne_q, silent_q ⟩ ⟩,
5    -- To get a contradiction, we need p and q as our truthy processors
6    have f_exists := third_exists p q,
7    cases f_exists with f hf,
8    use f, -- The faulty processor
9    let value_map : procs → ℕ := λ x, 1,
10   use value_map,
11   let σ : list procs → ℕ := λ l, truthy_like value_map l,
12   use σ,
13   intros l f_nin_l hσ,
14   rw [is_IC, is_interactive, is_consistent],
15   push_neg,
16   intro interactive,
17   exfalso,
18   have accurate := interactive p q ⟨ ne.symm hf.1, ne.symm hf.2 ⟩,
19   rw [silent_q] at accurate,
20   dsimp [map_list, value_map] at accurate,
21   rw [alg_sound] at accurate,
22   -- UPSHOT: When machines don't talk, interactivity can be broken
23   contradiction,
24 },
25 push_neg at W_full,
```

Figure 2: A proof of impossibility of interactivity in the case that two machines don't talk to each other.

- Lines 5 through 12 simply setup a trivial value map and networking protocol which we will use to break interactivity.
- Lines 18-23 The concluding lines take advantage of the soundness of the algorithm to show that the algorithm cannot guarantee interactivity.

References

[1] Leslie Lamport. "Reaching Agreement in the Presence of Faults". In: Journal of the ACM 27.2 (1980). DOI: https://dl.acm.org/doi/10.1145/322186.322188.

[2] Mark Pekala. Byzantine Verification. URL: https://github.com/mpekala23/har-ifvm-23-mpekala/tree/main/src/final.

4 Appendix: Code

```
import data.dfinsupp.multiset
import data.finset.basic
import data.finset.card
import data.finset.fin
import data.list.basic

/-
The goal of this file is to prove that in the case of three generals,
with one traitorous general, reaching consensus is impossible. After
completing this proof, we state the result of the general theorem, but
do not present a proof.
-/

/- SECTION ONE: Definitions -/

-- This codifies the requirement that machines actually talk to each other,
-- and do not just choose degenerate protocol
def is_interactive {fin : finset }
(wrong : fin)
(value_map : fin  )
(assignment : fin  fin  )
:=  p q : fin, p  wrong  q  wrong  assignment p q = value_map q

-- This codifies the requirement that the machines agree on _everything_.
-- Note that this includes the requirement that truthy machines agree on the
-- value of faulty machines, either meaning they both identify it as faulty,
-- or both receive the same value.
def is_consistent { fin : finset  }
(wrong : fin)
(value_map : fin  )
(assignment : fin  fin  )
:=  p q : fin, p  wrong  q  wrong   x : fin, assignment p x = assignment q x

-- This simply wraps the above two definitions into a handy statement
def is_IC {fin : finset }
(wrong : fin)
(value_map : fin  )
```

```
(assignment : fin  fin  )
:= is_interactive wrong value_map assignment  is_consistent wrong value_map
    ↪ assignment

-- A helper definition that defines what it means for a map to satisfy a
-- condition on the first element of a list (trivially true if empty)
def matches_first {fin : finset }
( : list fin  ) (value_map : fin  ) : list fin  Prop
| [] := true = true
| (h :: t) :=  (h :: t) = value_map h

-- A helper definition to capture the meaning of a list starting with a machine
def starts_with {fin : finset } (p : fin) : list fin  bool
| [] := true
| (h :: t) := h = p

-- A helper definition to capture the meaning of a list ending with a machine
def ends_with {fin : finset } (p : fin) : list fin  bool
| [] := false
| l := l.reverse.head' = some p

-- A simple result showing that the last element in a list must be unique
lemma end_unique {fin : finset } {a : fin} {b : fin}
(h : a  b)
(l : list fin)
(h_end : ends_with a l) : ends_with b l :=
begin
  cases l,
  {
    simp only [eq_ff_eq_not_eq_tt],
    rw ends_with,
    simp only [to_bool_false_eq_ff],
  },
  {
    rw ends_with at *,
    simp* at h_end,
    simp only [list.reverse_cons, bool.of_to_bool_iff],
    simp only [*, not_false_iff],
  },
end

-- Given a list of lists, return only the lists that start with q
def lists_that_start_with {fin : finset } (q : fin) : list (list fin)  list (list
    ↪  fin)
| [] := []
| ([] :: meta_l) := lists_that_start_with meta_l
| ((h :: rest) :: meta_l) :=
    if h = q
```

```
84      then (h :: rest) :: (lists_that_start_with meta_l)
85      else (lists_that_start_with meta_l)
86
87  -- Given a list of lists, return only the lists that end with q
88  def lists_that_end_with {fin : finset } (q : fin) : list (list fin)  list (list
        ↪ fin)
89  | [] := []
90  | ([] :: meta_l) := lists_that_end_with meta_l
91  | (l :: meta_l) :=
92      if l.last' = some q
93      then l :: (lists_that_end_with meta_l)
94      else (lists_that_end_with meta_l)
95
96  -- Combine the two above, and return the subset of a list of lists that start
97  -- with a given p and end with a given q
98  def filter_by {fin : finset } (p : fin) (q : fin) (W : list (list fin)) : list (
        ↪ list fin) :=
99  lists_that_end_with q (lists_that_start_with p W)
100
101 -- A result about our filter_by function that enforces the first element
        ↪ condition
102 -- NOTE: It is left unproven, but this should be possible to replace with a
        ↪ formal proof
103 lemma h_filter_by_start {fin : finset } {f : fin} {p : fin} {W : list (list fin)}
104 (l : list fin)
105 (hl : l  filter_by f p W) : starts_with f l := sorry
106
107 -- A result about our filter_by function that enforces the last element condition
108 -- NOTE: It is left unproven, but this should be possible to replace with a
        ↪ formal proof
109 def h_filter_by_end {fin : finset } {f : fin} {p : fin} {W : list (list fin)}
110 (l : list fin)
111 (hl : l  filter_by f p W) : ends_with p l := sorry
112
113 -- A helper function to return a list by applying a function to every
114 -- element of that list
115 def map_list { : Type*} { : Type*} (f :   ) : list   list
116 | [] := []
117 | (h :: t) := (f h)::(map_list t)
118
119 -- A helper function to generate truthy behavior when crafting network functions
120 def truthy_like {fin : finset } (value_map : fin  ) : list fin
121 | [] := 0
122 | (h :: t) := value_map h
123
124 /- SECTION TWO: Impossibility for n = 3 -/
125
126 -- These definitions are constant and provide the information at the global
```

```
127 -- level to restrict our scope to n = 3
128 constant procs : finset
129 constant procs_def : procs = { 1, 2, 3}
130 constant first_exists :  x : procs, true
131 constant second_exists (p : procs) :  x : procs, x  p
132 constant third_exists (p : procs) (q : procs) :  x : procs, x  p  x  q
133 constant procs_ne_zero :  p  procs, p  0
134
135 -- It is impossible for three generals with one traitor to agree on anything
136 theorem byzantine :
137  W : list (list procs), -- Any arbitrary communications
138  alg : list (list procs)  (list procs  )  , -- Any algorithm
139 alg [] = 0 -- The algorithm must be _sound_
140
141 -- The algorithm must be _blind_ (i.e., dependent on it's inputs)
142 ( K : list (list procs),   ' : list procs  ,  m:   , ( l  K, '(l) = m((l)))  alg
    ↪ K ' = m(alg K ))
143
144 -- We can always construct a counterexample
145  f : procs,
146  value_map : procs  ,
147   : list procs  ,
148  l : list procs, f  l  matches_first  value_map l
149
150 is_IC f value_map ( p q : procs, alg (filter_by q p W) ) :=
151 begin
152   -- Get access to the arbitrary protocl
153   intro W,
154   intro alg,
155   intro alg_sound,
156   intro alg_blind,
157
158   -- Handle the degenerate case where some machines don't talk to each other
159   by_cases W_full : ( p : procs,  q : procs, p  q  filter_by q p W = []),
160   {
161     rcases W_full with  p, q,  p_ne_q, silent_q  ,
162     -- To get a contradiction, we need p and q as our truthy processors
163     have f_exists := third_exists p q,
164     cases f_exists with f hf,
165     use f, -- The faulty processor
166     let value_map : procs   :=  x, 1,
167     use value_map,
168     let  : list procs   :=  l, truthy_like value_map l,
169     use ,
170     intros l f_nin_l h,
171     rw [is_IC, is_interactive, is_consistent],
172     push_neg,
173     intro interactive,
```

```
    exfalso,
    have accurate := interactive p q  ne.symm hf.1, ne.symm hf.2 ,
    rw [silent_q] at accurate,
    dsimp [map_list, value_map] at accurate,
    rw [alg_sound] at accurate,
    -- UPSHOT: When machines don't talk, interactivity can be broken
    contradiction,
  },
  push_neg at W_full,

  -- Get three distinct elements from procs, p q and f
  cases first_exists with p,
  cases second_exists p with q q_ne_p,
  rcases third_exists p q with  f,  f_ne_p, f_ne_q ,

  -- We can construct a counterexample with a simple value_map
  -- and arbitrary f
  let value_map : procs   :=  p, p,
  use [f, value_map],

  -- Assume that having a valid network function implies interactive consistency
  by_contradiction all_sigs,
  push_neg at all_sigs,

  -- A very simple (but valid) network function
  let  : list procs   :=  l,
    if starts_with f l
      then 1
      else truthy_like value_map l,

  -- A derivative (but also valid) network function that can trick the algorithm
  let ' : list procs   :=  l,
    if starts_with f l
      then if ends_with p l
        then  l + 1
      else if ends_with q l
        then  l + 2
        else truthy_like value_map l
      else truthy_like value_map l,

  -- By assumption, using this  will guarantee consistency
  rcases all_sigs  with  K, f_nin_K,  h, interactive, consistent  ,
  rw is_consistent at consistent,
  have alg_app_eq := consistent p q  ne.symm f_ne_p, ne.symm f_ne_q  f,

  -- By assumption, using this ' will guarantee consistency
  rcases all_sigs ' with  K, f_nin_K,  h', interactive', consistent'  ,
  rw is_consistent at consistent',
```

```
  have alg_app_eq' := consistent' p q  ne.symm f_ne_p, ne.symm f_ne_q  f,

  -- m and m' capture the discrepancies between  and '
  let m :    :=  n, n + 1,
  let m' :    :=  n, n + 2,

  -- Proofs that for the filtered values in question m and m' do indeed
  -- capture the differences between  and '
  have _rel_p :  (l : list procs), l  filter_by f p W  ' l = m ( l),
  {
    intros l hl,
    dsimp [m, , '],
    have l_starts_with_f := h_filter_by_start l hl,
    have l_ends_with_p := h_filter_by_end l hl,
    simp only [*, if_true, self_eq_add_left, nat.one_ne_zero],
  },
  have _rel_q :  (l : list procs), l  filter_by f q W  ' l = m' ( l),
  {
    intros l hl,
    dsimp [m', , '],
    have l_starts_with_f := h_filter_by_start l hl,
    have l_ends_with_q := h_filter_by_end l hl,
    have does_not_end_with_p := end_unique q_ne_p l l_ends_with_q,
    simp only [*, if_true, if_false],
  },

  -- Our algorithm is blind, so using these two network functions
  -- we can trick it
  have p'_eq_p_one := alg_blind (filter_by f p W)  ' m _rel_p,
  dsimp [m] at p'_eq_p_one,
  have q'_eq_q_two := alg_blind (filter_by f q W)  ' m' _rel_q,
  dsimp [m'] at q'_eq_q_two,

  -- Show that the equalities implied by blindness contradict each other
  rw alg_app_eq at p'_eq_p_one,
  rw alg_app_eq' at p'_eq_p_one,
  rw q'_eq_q_two at p'_eq_p_one,
  simp only [nat.succ_ne_self] at p'_eq_p_one,
  exact p'_eq_p_one,
end
```