Byzantine Fault Allowing Protocols

LIAM MONNINGER, Ramate LLC, USA

I've written this memo to describe to friends and colleagues the beginning of my study of a category of distributed computing protocols which I call Byzantine Fault Allowing. I begin with the motivation for the study. I then provide a trivial combinatorial example of a protocol assumed to be in the category. Finally, I suggest more abstract constructions with which the study is properly concerned.

Most terms and notation follow [HKR]. The final section introduces provisional and more suggestive terms.

1 Motivation

1.1 The General Need for Order

In computing, we typically reason about partial functions. These are functions which deterministically map one value to another, but may not be defined for all inputs.

$$\begin{aligned}&\hat{f} : S \to S \cup \{\bot\} \text{ is a lifted partial function.}\\ &f : S \rightharpoonup S \text{ is a partial function in its native category.}\end{aligned}$$

When composing these partial functions, we axiomatize that undefinedness propagates through the composition.

$$\begin{aligned}&f, g : S \rightharpoonup S\\ &f \circ g \text{ is defined if and only if}\\ &\quad g(x) \text{ is defined and}\\ &\quad f(g(x)) \text{ is defined}\end{aligned}$$

We find this to be a suitable model for modern machines, because we know how to design hardware which can logically represent functions with these properties and which is rarely corrupted by the surrounding environment.

However, for general computing tasks, we often have partial functions which are not commutative.

$$f \circ g \neq g \circ f$$

A familiar example is subtraction on $\mathbb{N}$, which is naturally partial:

$$\begin{aligned}&\text{sub} : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}\\ &\text{sub}(a, b) = \begin{cases} a - b & \text{if } a \geq b,\\ \text{undefined} & \text{otherwise.}\end{cases}\end{aligned}$$

Subtraction is also not commutative:

Author's Contact Information: Liam Monninger, liam@ramate.io, Ramate LLC, Durham, California, USA.

$$\text{sub}(5,3) = 2 \neq \text{sub}(3,5) \text{ (undefined)}.$$

In general, for a family $f_1, \ldots, f_n$ of partial functions, the composite is determined by the ordered sequence $(f_1, \ldots, f_n)$ not the unordered set $\{f_1, \ldots, f_n\}$. In local hardware cases–such as a single hart–we can often form reasonable assumptions or conventions, such that something like a state machine reading through an assembly is equivalent to a sequence of partial functions. In the context of general distributed computing tasks, however, we tend to make these assumptions more minimal.

In distributed computing, we may instead refer to a set of processes P which are able to share information with one another via messages $m \in M$ to varying degrees of success. Then, within this context, we hope to achieve results up to some notion of correctness and consistency, i.e., up to some notion of equivalence with the local hardware case.

Put more provocatively, a distributed computing protocol often has to decide from a chaotic world of messages what a reasonable order should be. Signatures can be placed on messages to ensure they are authentic. Proving systems can be built to attest the correctness of the value of a partial function. But, these may mean little without a reasonable order.

1.2 2f + 1

Acknowledgments