

Data failures (brief announcement)

Simona Orzan¹ and Mohammad Torabi Dashti²

¹ TU/e, The Netherlands

² ETH Zurich, Switzerland

Abstract. In an asynchronous environment where processes are prone to byzantine failures, algorithm design is an enormous complex and tedious task. To improve the theoretical understanding of the byzantine model and, hopefully, enable a clear modular design of algorithms, we propose to decompose the byzantine behaviour into a *data failure* behaviour and a *communication failure* behaviour. We argue that the two failure types are orthogonal and we point out how they generate a range of several new interesting failure models. Some of them, where communication is reliable but the local memories are prone to arbitrary data loss, are also relevant as environment abstractions for emerging security applications like smartcard protocols.

1 Byzantine failures decomposed

We characterize formally *the byzantine behaviour* by identifying orthogonal aspects of it. The goal is to allow studying the solvability of basic distributed computing problems in failure models which are less difficult than byzantine, but more complex or complementary to the already well understood crash model.

A formal algorithm model Let V be a set of variable names. Every process owns a local variable set $D \in V^*$, an input buffer ib , modelled as a list of messages $v \leftrightarrow i$ (value v sent by process i), and an output buffer ob , which is a list of messages $v \leftrightarrow i$ (value v has to be sent to process i). A *local algorithm* is a finite sequence of guarded actions:

$$\begin{aligned} \text{action} & ::= g \Rightarrow \text{in}(x, j) \mid g \Rightarrow \text{out}(v, i) \mid g \Rightarrow x := x' \mid g \Rightarrow \text{loop} \text{ actblock} \\ \text{actblock} & ::= \text{action} \mid \text{action}; \text{actblock} \end{aligned}$$

where g is a predicate on the data variables in D and the action guarded by g is only executed if g evaluates to true, otherwise skipped. $\text{in}(x, j)$ transforms the current local state $(D, ib + [v \leftrightarrow i], ob)$ into $(D[v/x, i/j], ib - [v \leftrightarrow i], ob)$, $\text{out}(v, i)$ transforms it into $(D, ib, ob + [v \leftrightarrow i])$ and $x := x'$ into $(D[x/x'], ib, ob)$. We omit details like data types, they can be implemented on top of the structure above. We consider, w.l.o.g., all variables of type Nat . These specifications generate, in the usual way, sets of traces or possible runs of the algorithms described. A *distributed algorithm* for N processes is a tuple $\mathcal{A} : \langle A_1 \dots A_N \rangle$ with A_i algorithms written in the language above. A *problem* is a tuple (\vec{x}, \vec{y}, Req) , with \vec{x} and \vec{y} the input and output vectors, respectively, and Req a list of consistent requirements on \vec{x}, \vec{y} expressed as LTL formulae. For a similar use of LTL, see [3].

Orthogonal failures We define two types of deviations from the expected behaviour: a *data failure* is inserting assignments $(\top \Rightarrow x := x')$ actions in an algorithm description; a *communication failure* is inserting or deleting one or more $\top \Rightarrow \text{in}$ or $\top \Rightarrow \text{out}$ actions in an algorithm description. These two aspects are mutually independent and their combination give rise to interesting failure models, as discussed further. When both types of failures are considered simultaneously, the full power of a byzantine model is reached:

COMMUNICATION ↓	DATA →	<i>no failure</i>	<i>reset failure</i>	<i>edit failure</i>
<i>no failure</i>		ideal sync	RF	DF
<i>in/out permanently ignored</i>		crash	RCF	DCF
<i>some in/out ignored</i>		crash-recovery	RCRF	DCRF
<i>some in/out ignored or/and added</i>		random-com	RRCF	BYZ

Table 1. A taxonomy of failure models. The bold ones involve various degrees of data corruption and are not yet studied in the literature

Theorem 1 (byzantine decomposition). *Any byzantine behaviour can be simulated by a combination of data and communication failures.*

A map of failure models Well-known failure models fit in our two-dimensional framework: the *crash* failure is a communication failure where, from one point on, all *out* actions are ignored; the *general omission* or *crash-recovery* failure is a communication failure where some of the *in* and *out* actions are ignored; the *muteness* failure, as in [2, 1] is a communication failure where, from one point on, all *in* and *out* actions are ignored, except actions of the form $\text{out}(\text{heartbeat}, H)$ where H is a process implementing a failure detector. These are all concerned with communication failures, but do not consider any form of data tempering. In order to shed new light on the origin of byzantine difficulties, we extend the collection of intermediate failure models with a new dimension, where two degrees of data corruption are considered: *reset* failures, meaning random addition of $x := \lambda$ actions, with λ a default value; and *edit* failures, meaning random addition of $x := x'$ actions. Table 1 gives an overview. We think the most interesting of these models are the *pure reset failure (RF)*, the *data failure (DF)* and the *data-crash failure (DCF)* models. DF is discussed separately below, the other two are left for further study. The reset failures can be mostly reduced to crash failures, as shown by the following theorem:

Theorem 2 (reset). *Let M be a communication failure model allowing crashing and \mathcal{A} a correct distributed algorithm in M . If λ is a globally known value, then \mathcal{A} can be transformed in \mathcal{A}' , which is correct in the $(M + \text{reset failures})$ model.*

2 DF, the pure data failure model

In DF, processes are prone to data failures, but communication works flawlessly. This allows designing of detection mechanisms to deal specifically with data corruption. Hopefully, some of them can later be adapted for more complex failure environments. DF is also a realistic failure model for security applications. Malicious overwriting of own data, like the saldo on the bank account, while not interfering with the algorithm running on the card, fits in this model. Also physical attacks on trusted devices like SIM cards or bankcards might have as effect the corruption of data, although the card will look like it continues to function.

It is well-known that consensus is impossible to achieve when processes follow their algorithm, but may crash. We show that this impossibility also holds in the complement model, when all processes are crash-free but their data is susceptible to errors.

Theorem 3 (general impossibility). *Let $P = (\vec{x}, \vec{y}, \text{Req})$ be a non-trivial problem, i.e. at least one formula in Req depends on \vec{x} and/or \vec{y} . In the pure data failure model, there exist no terminating distributed algorithm \mathcal{A} to solve problem P .*

This means that also in superseeding models, in particular in the byzantine model, problems cannot be solved without extra assumptions or mechanisms. Indeed, the classical solutions for multiparty computation problems rely on the famous $t < n/3$ assumption and several works on distributed consensus use byzantine failure detectors [1].

Data failure detection Algorithm design for communication failure environments essentially relies on failure detectors to enforce some level of synchrony. In DF, we are looking for similar mechanisms that could enforce some level of data reliability. We take inspiration from the security protocols, where encryption, check-sums, verification functions, etc. are used for the similar purpose of protecting data.

A *corruption pattern* is a function $CP : \Omega \rightarrow 2^V$ mapping time t to the set of variables that got corrupted until time t in all local memories. A *message verifier* is a function $MV : \mathcal{P}roc \times \Omega \rightarrow 2^{Msg}$, with $MV(p, t) = \{m_1 \dots m_k\}$ being a set of corrupted messages received by p until moment t . A *process verifier* is a function $PV : \mathcal{P}roc \times \Omega \rightarrow 2^{\mathcal{P}roc}$, with $PV(p, t) = \{p_1 \dots p_k\}$ being a set of processes that own at least one corrupted variable until moment t . The perfect verifier always returns to p the whole set of corrupted messages or the whole set of corrupted processes, respectively. Weaker verifiers can be defined, following the model of communication failure detectors.

A different approach, which seems natural when data is involved, is to prevent – rather than detect, the failures. An abstraction of this idea are *protected variables*: each process is equipped with a set of variables that can not be subject to edit failures. A possible way to implement protected variables is using encryption and digital signatures. Of course, this would only protect against malicious processes and not against physical attacks.

Using data failure detection, the impossibility of Theorem 3 can be circumvented. For example, the following holds:

Theorem 4 (DC solvable with a verifier and protected variables). *In DF, two-party distributed consensus is solvable in the presence of a perfect message verifier and a variable protection mechanism.*

3 Discussion

Trade-offs in the DF model Obviously, if all variables are protected, then the model becomes a failure-free asynchronous network and trivial algorithms will work. But this is an expensive solution, so an interesting question is whether some of the protection could be replaced by smart optimistic algorithm designs - that would be able to detect corruption of unprotected variables, if needed, and would be fast in the no failure case.

Program counters may fail? Defining the borders of what a data failure should be is tricky. In realistic situations, any part of the memory might fail, including program counters and even the program code. In order to keep the model simple (allowing tempering with program counters can transform any program with at least one out primitive in a byzantine program), we choose an abstraction level where these are regarded as belonging to a trusted and reliable operating system, and only explicit program variables are exposed to failures.

Algorithm combination We identified failure models which are less complex than the byzantine model. A natural, although ambitious question is if/when algorithm combination is

possible: given an algorithm \mathcal{A} using data failure detectors that solves P in DF and an algorithm \mathcal{B} using crash failure detectors that solves P in CF, is there a way to build an algorithm \mathcal{C} that solves P in DF+CF?

References

1. A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: from crash to byzantine failures. In *Ada-Europe*, volume 2361 of *LNCIS*, 2002.
2. Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *Comput. J.*, 46(1):16–35, 2003.
3. S.M. Orzan and M. Torabi Dashti. Fair exchange and distributed consensus are incomparable, 2008. <http://www.win.tue.nl/~sorzan/papers/fair-exchange.pdf>.

APPENDIX: proof ideas

Theorem 1 A byzantine process is, in our formal model, a random set of traces. The operations allowed by the two types of failures are adding assignments and deleting/adding in and out actions. Starting from a given “well-behaved” trace, these operations are enough to generate any byzantine trace — modulo the original assignments that can not be deleted, but can be made ineffective. \square

Note that none of the two operations can achieve the same transformation alone. Pure data failures can trigger generation of new in/out operations only under very specific assumptions on the original well-behaved algorithm, so they can not lead in general to a simulation of a byzantine trace. Pure communication failures can generate arbitrarily long traces of visible (out) behaviour, but the messages being sent cannot be made up, they are just copies of messages provided by the original algorithm.

Theorem 2 Eliminate the use of λ in \mathcal{A} , by replacing λ with a new, not yet used, constant from Nat . Replace every action $g \Rightarrow \text{out}(x, i)$ with the sequence $g \wedge (x \neq \lambda) \Rightarrow \text{out}(x, i); (x = i) \Rightarrow \text{crashstop}$. \square

Theorem 3 Suppose \mathcal{A} exists. Pick formula $\phi \neq \top$ on \vec{x}, \vec{y} from the *Req* list. Then there exists a valuation for \vec{x}, \vec{y} s.t. $\neg\phi$ is satisfied. At the end of \mathcal{A} , assign to variables \vec{x} and \vec{y} that valuation. \square

Theorem 4 We write \boxed{x} to denote that x is a protected variable. The local algorithm for process p_1 is A_1 :

$$\begin{aligned} \boxed{j} &:= 2; \\ \top &\Rightarrow \text{out}(x_1, p_1); \\ \top &\Rightarrow \text{in}(m, j); \\ m \notin MV(p_1, t) &\Rightarrow \boxed{y_1} := \min(x_1, m); \end{aligned}$$

A_2 for p_2 is similar. The distributed algorithm $\langle A_1, A_2 \rangle$ solves the two-party consensus in DF with a perfect verifier and two protected variables. \square

A remark can be made here: In the spirit of classical distributed algorithms, we consider the output of failing parties irrelevant; the requirements only refer to correct parties. However, if the focus moves to security aspects, the failing parties do matter - either

because they are honest and should be protected against the external attacker, or because they are dishonest and the other parties should be protected (in contract signing, for instance, the failing parties should not get a valid signature as output!). We think that this difference can simply be captured by new security-oriented requirements and that the underlying failure models and the failure detector solutions are the same.