Distributed Computing in the Presence of Mobile Faults

Nicola Santoro* Peter Widmayer†

1 Introduction

1.1 Distributed Computing and Communication Faults

A *distributed computing environment* is a collection of networked computational *entities* communicating with each other by means of *messages*, in order to achieve a common goal; e.g., to perform a given task, to compute the solution to a problem, to satisfy a request either from the user (i.e., outside the environment) or from other entities. Although each entity is capable of performing computations, it is the collection of all these entities that together will solve the problem or ensure that the task is performed. Such are for example distributed systems, grids of processors, data communication networks, distributed databases, transaction processing systems. More precisely, our universe is a system of $n \geq 2$ entities $x_1, \ldots, x_n$ connected through dedicated communication links. The connection structure is an arbitrary undirected, connected simple graph $G = (V, E)$. We will denote by $deg(G)$ the maximum node degree in G, by $con(G)$ the edge-connectivity of G, by $diam(G)$ the diameter of G, and by $m(G)$ the number of links of G.

In a distributed computing environment, the entities behave according to a specified set of rules, called *distributed algorithm* or *protocol* , that specify what each entity has to do. The collective but autonomous execution of those rules enables the entities to perform the desired task, to solve the problem. The process of designing a correct set of rules which will correctly solve the problem with a small cost is rather complex (e.g., [**?**]). In all non-trivial computations, the entities need to communicate, and they do so by sending messages to (and receiving messages from) their neighbours in the underlying communication network G. The message sent by an entity need not be the same for all destination entities, i.e., separate links allow for different messages to different destinations at the same time.

A major obstacle to distributed computations is the possibility of *communication faults* in the system. A *communication* is a pair (α, β) of messages $\alpha, \beta \in M \cup \{\Omega\}$ for a pair (x, y) of neighbouring entities called *source* and *destination*, where M is a fixed and possibly infinite message universe and Ω is the null message: α is the message sent by the source and β is the message received by the destination; by convention $\alpha = \Omega$ denotes that no message

*School of Computer Science, Carleton University
†Institute for Theoretical Informatics, ETH Zurich

is sent, and $\beta = \Omega$ denotes that no message is received. A communication (α, β) is *faulty* if $\alpha \neq \beta$, non-faulty otherwise. There are three possible types of *communication faults*:

1. An *omission*: the message sent by the source is never delivered to the destination $(\alpha \neq \beta = \Omega)$.

2. An *addition*: a message is delivered to a destination although none was sent by the source $(\Omega = \alpha \neq \beta)$.

3. A *corruption*: a message is sent by the source but one with different content is received by the destination $(\Omega \neq \alpha \neq \beta \neq \Omega)$.

While the nature of omissions and corruptions is quite obvious, that of additions is less so. Indeed, it describes a variety of situations. The most obvious one is when sudden noise in the transmission channel is mistaken for transmission of information by the neighbour at the other end of the link. The more important occurrence of an addition is rather subtle: it describes the reception of a "non-authorized message" , i.e., a message that appears to come from the source, but does not have its consent. In other words, additions represent messages surreptitiously inserted in the system by some outside, and possibly malicious, entity. Spam being transmitted from an unsuspecting network site clearly fits the description of an addition. Summarizing, additions do occur and can be very dangerous.

Notice that other types of failures can be easily modeled by communication faults; for instance, omission of all messages sent by and to an entity can be used to describe the crash failure of that entity. Analogously, it is possible to model the intermittent failure of a link in terms of occasional omissions in the communication between the two entities connected by the link. In fact, most processor and link failure models can be seen as a special *localized* case of the communication failure model, where all the faults are restricted to the communications involving a fixed (though, a priori unknown) set of entities or of links.

1.2 Fault-Tolerant Computations and Mobile Faults

Not all communication faults lead (immediately) to computational errors (i.e., to incorrect results of the protocol), but some do. So the goal is to achieve *fault-tolerant* computations; that is, the aim is to design protocols that will proceed correctly in spite of the faults. Clearly no protocol can be resilient to an arbitrary number of faults. In particular, if the entire system collapses, no protocol can be correct. Hence the goal is to design protocols that are able to withstand up to a certain amount of faults of a given type.

Another fact to consider is that not all faults are equally *dangerous*. The danger of a fault lies not necessarily in the severity of the fault itself but rather in the consequences that its occurrence might have on the correct functioning of the system. In particular, danger for the system is intrinsically related to the notion of detectability. In general, if a fault is easily detected, a remedial action can be taken to limit or circumvent the damage; if a fault is hard or impossible to detect, the effects of the initial fault may spread throughout the network creating possibly irreversible damage.

An important distinction is whether the faults are localized or mobile.

- *Localized* faults occur always in the same region of the system; that is, only a fixed (although a priori unknown) set of entities/links will exhibit a faulty behaviour.
- *Mobile* faults are not restricted to a fixed (but a priori unknown) set of links, but can occur between any two neighbours [**?**].

Notice that, in general, the majority of failures have mostly a transient and ubiquitous nature; that is, faults can occur anywhere in the system and, following a failure, normal functioning can resume after a finite (although unpredictable) time. In particular, failures will occur on any communication link, almost every entity will experience at one time or another a send or receive failure, etc. Mobile faults are clearly more difficult to handle than the ones that occur always in the same places. In the latter case, once a fault is detected, we know that we can not trust that link; with mobile faults, detection will not help us with the future events.

With this in mind, when we talk about fault-tolerant protocols and fault-resilient computations we must always qualify the statements and clearly specify the type and number of faults that can be tolerated.

Unfortunately, the presence of communication faults renders the solution of problems difficult if not impossible. In particular, in *asynchronous* settings, the mere possibility of *omissions* renders all non trivial tasks unsolvable, even if the faults are *localized* to (i.e., restricted to occur on the links of) a single entity [**?**]. Due to this impossibility result, the focus is on *synchronous* environments.

Since synchrony provides a perfect omission detection mechanism [**?**], localized omissions are easily dealt with in these systems. Indeed *localized* faults in synchronous environments have been extensively investigated in the *processor failure* model mostly for complete networks (e.g., see [**?**, **?**, **?**, **?**, **?**, **?**]), as well as in the *link failure* model and in the *hybrid failure* models that consider both links and entities (e.g. [**?**, **?**, **?**]).

The immediate question is then whether synchrony allows to tolerate also *mobile* communication faults; that is, faults that are not restricted to a fixed (but a priori unknown) set of links, but can occur between any two neighbours.

In this chapter we will examine how we can deal with these communication failures, also called *dynamic faults* or *ubiquitous faults* in synchronous distributed computing systems. It is not surprising that the number of dynamic faults that can be tolerated at each time unit is by far less than that of the localized and permanent faults we can deal with. What is surprising is perhaps the fact that something *can* be done at all.

2 The Boundaries of Computability

The goal of this section is to examine the boundaries of computability in a synchronous distributed environment where communication faults are ubiquitous.

Defining what is computable in the presence of mobile faults seems to be a very arduous task. Like in the case of localized faults, we will focus on a very basic problem, *agreement* , and study under what conditions this problem is solvable, if at all. The basic nature of the

agreement problem is such that, if it can not be solved under a set of conditions on the nature and amount of faults, then *no* non-trivial problem can be solved under those conditions. Determining under what conditions agreement can be achieved provides a threshold for the computability of all other non-trivial tasks and functions.

2.1 Agreement, Majority, and Unanimity

Each entity x_i has an input register with an initial value r_i, and an output register for which it must choose a value v_i as the result of its computation. For simplicity, we limit ourselves to Boolean inputs, i.e., $r_i \in \{0, 1\}$ for all i; all the results hold also for non-binary values. In the *K-agreement problem* (**Agree(K)**), at least K entities must terminally choose the same *decision* value $v \in \{0, 1\}$ within a finite amount of time, subject to the *validity* constraint that, if all input values r_i were the same, then v must be that value. Here, "terminally" means that, once made, the decision can not be modified.

Depending on the value of parameter K, we have different types of agreement problems. Of particular interest are

- $K = \lceil \frac{n}{2} \rceil + 1$ called *strong majority*;
- $K = n$ called *unanimity* (or *consensus*), in which all entities must decide on the same value.

Note that any agreement requiring *less* than a strong majority (i.e., $K \leq \lceil n/2 \rceil$) can be trivially reached without any communication; e.g., each r_i chooses $v = r_i$. We are interested only in *non-trivial* agreements (i.e., $K > \lceil n/2 \rceil$).

A possible solution to the unanimity problem is, for example, having the entities compute a specified Boolean function (eg., **AND** or **OR**) of the input values and choose its result as the decision value; another possible solution is for the entities to first elect a leader and then choose the input value of the leader as the decision value. In other words, unanimity (fault-tolerant or not) can be solved by solving any of a variety of other problems (e.g., function evaluation, leader election, etc). For this reason, if the consensus problem cannot be solved, then none of those other problems can.

2.2 Impossibility

Let us examine how much more difficult it is to reach a non-trivial (i.e., $K > \lceil \frac{n}{2} \rceil$) agreement in the presence of dynamic communication faults.

Consider for example a d-dimensional *hypercube*. From the results established in the case of entity failures, we know that if only one entity crashes, the other $n - 1$ can agree on the same value [**?**]. Observe that with d *omissions* per clock cycle, we can simulate the "send failure" of a single entity: all messages sent from that entity are omitted at each time unit. This means that, if d omissions per clock cycle are localized to the messages sent by the same single entity all the time, then agreement among $n - 1$ entities is possible. What happens

if those d omissions per clock cycle are *mobile* (i.e., not localized to the same entity all the time) ?

Even in this case, at most a single entity will be isolated from the rest at any one time; thus, one might still reasonably expect that an agreement among $n-1$ entities can be reached even if the faults are dynamic. Not only this expectation is false, but actually it is impossible to reach even strong majority (i.e., an agreement among $\lceil n/2 \rceil + 1$ entities).

If the communication faults are arbitrary (the *Byzantine* case), the gap between the localized and mobile cases is even stronger. In fact, in the hypercube, an agreement among $n-1$ entities is possible in spite of d Byzantine communication faults if they are localized to the messages sent by a single entity [**?**]; on the other hand, if the Byzantine faults are mobile then strong majority is impossible even if the number of faults is just $\lceil d/2 \rceil$.

These results for the hypercube are an instance of more general results by Santoro and Widmayer [**?**] that we are going to discuss next. Let **Omit**, **Add**, and **Corr** denote a system when the communication faults are only omissions, only additions, and only corruptions, respectively. We will denote by **X+Y** a system where the communication faults can be both of type **X** or **Y**, in an arbitrary mixture.

The first result is that, with $deg(G)$ omissions per clock cycle, strong majority cannot be reached:

Theorem 1 [**?**] *In* **Omit**, *no K-agreement protocol $\mathcal{P}$ is correct in spite of $deg(G)$ mobile communication faults per time unit if $K > \lceil n/2 \rceil$.*

If the failures are any mixture of corruptions and additions, the same bound $deg(G)$ holds for the impossibility of strong majority:

Theorem 2 [**?**] *In* **Add+Corr**, *no K-agreement protocol $\mathcal{P}$ is correct in spite of $deg(G)$ mobile communication faults per time unit if $K > \lceil n/2 \rceil$.*

In the case of arbitrary faults (omissions, additions, and corruptions: the Byzantine case), strong majority cannot be reached if just $\lceil deg(G)/2 \rceil$ communications may be faulty:

Theorem 3 [**?**] *In* **Omit+Add+Corr**, *no K-agreement protocol $\mathcal{P}$ is correct in spite of $\lceil deg(G)/2 \rceil$ mobile communication faults per time unit if $K > \lceil n/2 \rceil$.*

These results, summarized in Figure 1, are established using the proof structure for dynamic faults introduced in [**?**]. Although based on the bivalency argument of Fischer, Lynch and Paterson [**?**], the framework differs significantly from the ones for asynchronous systems since we are dealing with a *fully synchronous* system where time is a direct computational element, with all its consequences; e.g., non-delivery of an expected message is detectable, unlike asynchronous systems where a "slow" message is indistinguishable from an omission; etc. This framework has been first defined and used in [**?**]; more recently, similar frameworks have been used also in [**?**, **?**, **?**].

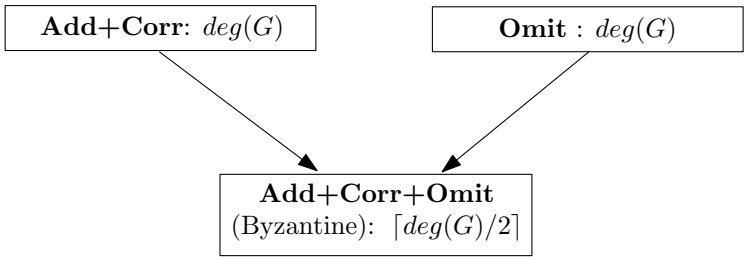

Figure 1: Minimum number of dynamic communication faults per clock cycle that may render strong majority impossible.

2.3 Possibility

Let us examine now when agreement among the entities is possible in spite of dynamic faults. We are interested in determining the maximum number F of dynamic communication faults of a given type that can be tolerated when achieving a non-trivial agreement.

Surprisingly, even *unanimity* can be achieved in several cases (a summary is shown in Figure 2). In these cases, we will reach unanimity, in spite of F communication faults per clock cycle, by computing the **OR** of the input values and deciding on that value. This is achieved by first constructing (if not already available) a mechanism for correctly broadcasting the value of a bit within a fixed amount of time T in spite of F communication faults per clock cycle. This reliable broadcast, once constructed, is then used to correctly compute the logical **OR** of the input values: all entities with input value 1 will reliably broadcast their value; if at least one of the input values is 1 (thus, the result of **OR** is 1), then everybody will be communicated this fact within time T; on the other hand, if all input values are 0 (thus, the result of **OR** is 0), there will be no broadcasts and everybody will be aware of this fact within time T. The variable T will be called *timeout*. The actual reliable broadcast mechanism as well as the value of F will obviously depend on the nature of the faults.

All the results of this section are from [**?**].

2.3.1 Single Type Faults

Let us first consider systems with a single type of dynamic communication faults. Some of the results are sometimes counterintuitive.

Corruptions
If the faults are just *corruptions*, unanimity can be reached *regardless of the number of faults*:

Theorem 4 *In* **Corr**, *regardless of the number of mobile faults, unanimity can be reached in time* $T = diam(G)$ *transmitting at most* $2\ m(G)$ *bits.*

To understand this result, first consider that, since the only faults are corruptions, there are no omissions; thus, any transmitted message will arrive, although its content may be corrupted. This means that if an entity starts a broadcast protocol, every node will receive a message (although not necessarily the correct one). We can use this fact in computing the **OR** of the input values. Each entity with a input value 1 starts a broadcast. Regardless of its content, a message will always and only communicate the existence of an initial value 1. An entity receiving a message thus knows that the correct value is 1 regardless of the content of the message, and will forward it to all is neighbours (if it has not already done so). Each entity needs to participate in this computation (i.e., transmit to its neighbours) at most once. If there is an initial value 1, since there are no omissions, all entities will receive a message within time $T(G) = diam(G)$. If all initial values are 0, no broadcast is started and, since there are no additions, no messages are received; thus, all entities will detect this situation since they will not receive any message by time $T(G)$.

Additions
The same level of fault-tolerance holds also in systems where the faults are only *additions*, although the solution is more expensive. Indeed, if each entity transmits to its neighbours at each clock cycle, it leaves no room for additions. Hence the entities can correctly compute the **OR** using a simple diffusion mechanism in which each entity transmits for the first $T(G) - 1$ time units: initially, an entity sends its value; if during this time it is aware of the existence of a 1 in the system, it will only send 1 from that moment on. The process clearly can terminate after $T(G) = diam(G)$ clock cycles. Hence

Theorem 5 *In* **Add**, *regardless of the number of mobile faults, unanimity can be reached in time* $T = diam(G)$ *transmitting at most* $2m(G)\ diam(G)$ *bits.*

Observe that if a spanning-tree $\mathcal{T}(G)$ of G is available, it can be used for the entire computation. In this case, the number of bits is $2(n-1)\ diam(\mathcal{T}(G))$ while time is $diam(\mathcal{T}(G))$.

Omissions
Consider the case when the communication errors are just *omissions*. Let $F \leq con(G) - 1$. When broadcasting in this situation, it is rather easy to circumvent the loss of messages. In fact, it suffices that the initiator of the broadcast, starting at time 0, continues transmitting it to all its neighbours in each time step until time $T(G) - 1$ (the actual value of the timeout $T(G)$ will be determined later); an entity receiving the message at time $t < T(G)$ transmits the message to all its other neighbours in each time step until time $T(G) - 1$. Call this process *Bcast-Omit* .

It is not difficult to verify that for $T(G)$ large enough (e.g., $T(G) \geq con(G)\ (n-2)\ +1$), the broadcast succeeds in spite of $f \leq con(G) - 1$ omissions per clock cycle. Let us denote by $T^*(G)$ the *minimum* timeout value ensuring that the broadcast is correctly performed in G in spite of $con(G) - 1$ omissions per clock cycle. Then, using *Bcast-Omit* to compute the **OR** we have:

Theorem 6 *In* **Omit**, *unanimity can be reached in spite of* $F = con(G) - 1$ *mobile communication faults per clock cycle in time* $T = T^*(G)$ *transmitting at most* $2m(G)\ T^*(G)$ *bits.*

We will discuss the estimates on the actual value of $T^*(G)$ later in the chapter, when discussing broadcast.

2.3.2 Composite Types of Faults

Let us consider now systems where more than one type of mobile faults can occur. In each graph G there are at least $con(G)$ edge-disjoint paths between any pair of nodes. This fact has been used in the component failure model to show that, with enough redundant communications, information can be correctly propagated in spite of faults and that the entities can reach some form of agreement.

Omissions and Corruptions

If the system suffers from *omissions and corruptions*, the situation is fortunately no worse than that of systems with only omissions. Since there are no additions, no unintended message is received. Indeed, in the computation of the **OR**, the only intended messages are those originated by entities with initial value 1 and only those messages (possibly corrupted) will be transmitted along the network. Thus, an entity receiving a message knows that the correct value is 1, regardless of the content of the message. If we use the same mechanism we did for **Omit**, we are guaranteed that everybody will receive a message (regardless of its content) within $T = T^*(G)$ clock cycles in spite of $con(G) - 1$ or fewer omissions, if and only if at least one originated (i.e., if there is at least one entity with initial value 1). Hence

Theorem 7 *In* **Corr+Omit**, *unanimity can be reached in spite of* $F = con(G) - 1$ *faults per clock cycle. The time to agreement is* $T = T^*(G)$ *and the number of bits is at most* $2\ m(G)\ T^*(G)$.

Omissions and Additions

In the case of systems with *omissions and additions*, consider the following strategy. To counter the negative effect of additions, each entity transmits to all its neighbours in every clock cycle. Initially, an entity sends its value; if at any time it is aware of the existence of a 1 in the system, it will send only 1 from that moment on. Since there are no corruptions, the content of a message can be trusted. Clearly, with such a strategy, no additions can ever take place. Thus, the only negative effects are due to omissions; however, if $F \leq con(G) - 1$, omissions can not stop the nodes from receiving a 1 within $T = T^*(G)$ clock cycles if at least one entity has such an initial value. Hence

Theorem 8 *In* **Add+Omit**, *unanimity can be reached in spite of* $F = con(G) - 1$ *mobile communication faults per clock cycle. The time to agreement is* $T = T^*(G)$ *and the number of bits is at most* $2\ m(G)\ T^*(G)$.

Additions and Corruptions

Consider the environment when faults can be both *additions and corruptions*. In this environment messages are not lost but none can be trusted; in fact the content could be incorrect (i.e., a corruption) or it could be a fake (i.e., an addition). This makes the computation of **OR** quite difficult. If we only transmit when we have 1 (as we did with only *corruptions*), how can we trust that a received message was really transmitted and not caused by an addition ? If we always transmit the **OR** of what we have and receive (as we did with only *additions*), how can we trust that a received 1 was not really a 0 transformed by a corruption ?

For this environment, indeed we need a more complex mechanism employing several techniques, as well as knowledge of the network G by the entities. The first technique we use is that of *time slicing* [**?**], in which entities are going to propagate 1 only at odd ticks and 0 at even ticks.

Technique *Time Slice*:

1. Distinguish between *even* and *odd* clock ticks; an even clock tick and its successive odd tick constitute a *communication cycle*.
2. To broadcast 0 (resp. 1), x will send a message to all its neighbours only on *even* (resp., *odd*) clock ticks.
3. When receiving a message at an *even* (resp., *odd*) clock tick, entity y will forward it only on *even* (resp., *odd*) clock ticks.

This technique however does not solve the problem created by additions; in fact, the arrival of a fake message created by an addition at an odd clock tick can generate an unwanted propagation of 1 in the system through the odd clock ticks. To cope with the presence of additions, we use another technique based on the edge-connectivity of the network. Consider an entity x and a neighbour y. Let $SP(x, y)$ be the set of the $con(G)$ shortest disjoint paths from x to y, including the direct link (x, y). To communicate a message from x to y, we use a technique in which the message is sent by x simultaneously on all the paths in $SP(x, y)$. This technique, called *Reliable Neighbour Transmission*, is as follows:

Technique *Reliable Neighbour Transmission*

1. For each pair of adjacent entities x, y and paths $SP(x, y)$, every entity determines in which of these paths it resides.
2. To communicate a message M to neighbour y, x will send along each of the $con(G)$ paths in $SP(x, y)$ a message, containing M and the information about the path, for t consecutive communication cycles (the value of t will be discussed later).
3. An entity z on one of those paths, upon receiving in communication cycle i a message for y with the correct path information, will forward it only along that path for $t - i$ communication cycles. A message with incorrect path information will be discarded.

Note that incorrect path information (due to corruptions and/or additions) in a message for y received by z is *detectable* and so is incorrect timing since (1) because of local orientation, z knows the neighbour w from which it receives the message; (2) z can determine if w is really its predecessor in the claimed path to y; and (3) z knows at what time such a message should arrive if really originated by x.

Let us now combine these two techniques. To compute the **OR**, all entities broadcast their input value using the *Time Slice* technique: the broadcast of 1's will take place at odd clock ticks, that of 0's at even ones. However, every step of the broadcast, in which every involved entity sends the bit to its neighbours, is done using the *Reliable Neighbour Transmission* technique. This means that each step of the broadcast now takes t communication cycles.

A corruption can now have one of two possible effects: Either it corrupts the path information in a message and causes the message not to reach its destination (regardless of whether the content of the message is correct or also corrupted), or it merely corrupts the content of the message, but leaves the path information correct. In the former case, the corruption will act like an omission (due to the way messages travel along paths), while in the latter, the message will arrive, and the clock cycle in which it arrives at y will indicate the correct value of the bit (even cycles for 0, odd for 1). Therefore, if x transmits a bit and the bit is not lost, y will eventually receive one and be able to decide the correct bit value. This is however not sufficient. We need now to choose the appropriate value of t so that y will not mistakenly interpret the arrival of bits due to additions, and will be able to decide if they were really originated by x. This value will depend on the length l of a longest one of the $con(G)$ shortest paths between any two entities. If no faults occur, then in each of the first $l-1$ communication cycles, y will receive a message through its direct link to x. In each of the following $t-l+1$ communication cycles, y will receive a message through each of its $con(G)$ incoming paths according to the reliable neighbour transmission protocol. Communication faults cause up to $f_c t$ messages to be lost during these t communication cycles (due to corruptions of path information), and cause up to $f_a t$ incorrect messages to come in, due to additions, with $f_c t + f_a t \leq (con(G)-1)t$. Because the number of incoming correct messages must be larger than the number if incoming incorrect messages, we get:

Lemma 1 *After $t > (con(G)-1)(l-1)$ communication cycles, y can determine the bit that x has sent to y.*

Consider that broadcast requires $diam(G)$ steps, each requiring t communication cycles, each composed of two clock ticks. Hence

Lemma 2 *It is possible to compute the* **OR** *of the input value in spite of $con(G)-1$ additions and corruptions in time at most $2diam(G)\ (con(G)-1)(l-1)$.*

Hence, unanimity can be guaranteed if at most $con(G)-1$ additions and corruptions occur in the system:

Theorem 9 *In* **Add+Corr**, *unanimity can be reached in spite of $F = con(G)-1$ mobile communication faults per clock cycle; the time is $T \leq 2\ diam(G)\ (con(G)-1)\ (l-1)$ and the number of bits is at most $4m(G)(con(G)-1)(l-1)$ messages.*

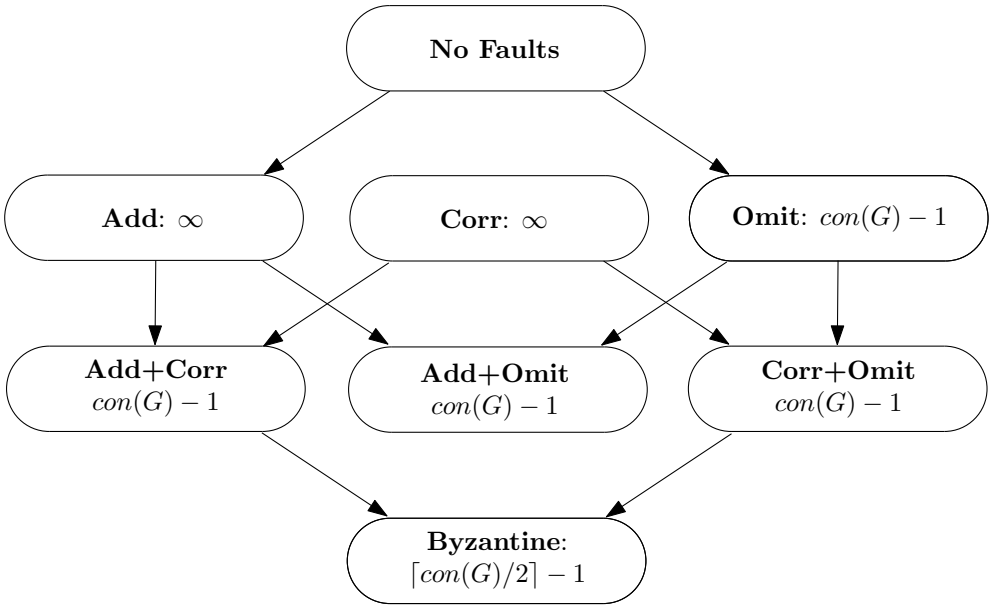

Figure 2: Maximum number of faults per clock cycle in spite of which *unanimity* is possible.

Byzantine Faults

In case of *Byzantine* faults, any type of faults can occur: omissions, additions and corruptions. Still, using a simpler mechanism than that for additions and corruptions we are able to achieve consensus, albeit tolerating fewer ($F = \lceil con(G)/2 \rceil - 1$) faults per clock cycle. To broadcast, we use precisely the technique *Reliable Neighbour Transmission* introduced to deal with additions and corruptions; we do *not* however use time slicing: this time, a communication cycle lasts only one clock cycle; that is, any received message is forwarded along the path immediately. The decision process (i.e., how y, out of the possibly conflicting received messages, determines the correct content of the bit) is according to the following simple Acceptance Rule: *y selects as correct the bit value received most often during the t time units.*

The technique *Reliable Neighbour Transmission* with this *Acceptance Rule* will work. Indeed, by choosing $t > (con(G) - 1)(l - 1)$, communication to a neighbour tolerates $\lceil con(G)/2 \rceil - 1$ Byzantine communication faults per clock cycle, and uses $(con(G) - 1)(l - 1) + 1$ clock cycles. Since broadcasting requires $diam(G)$ rounds of Reliable Neighbour Transmission, we have

Theorem 10 *In* **Omit+Add+Corr***, unanimity can be reached in spite of $\lceil con(G)/2 \rceil - 1$ mobile communication faults per clock cycle in time $T \leq diam(G)\ ((con(G) - 1)\ (l - 1) + 1)$.*

2.4 Tightness of Bounds

For all systems, except those where faults are just corruptions or just additions (and in which unanimity is possible regardless of the number of mobile faults), the bounds we have discussed are similar except that the possibility bounds are expressed in terms of the *connectivity* $con(G)$ of the graph, while the impossibility bounds are in terms of the *degree* $deg(G)$ of the graph.

This means that, in the case of $deg(G)$-*edge-connected graphs*, the impossibility bounds are indeed tight:
(1) with the number of faults (or more) specified by the impossibility bound, even strong majority is impossible;
(2) with one less fault than specified by the impossibility bound, even unanimity can be reached, and
(3) any agreement among less than a strong majority of the entities can be reached without any communication.
In other words, in these networks, agreement is either trivial or complete or impossible. This large class of networks includes *hypercubes*, *toruses*, *rings*, *complete graphs*, etc. In these networks, there is a precise and complete "impossibility map" for the agreement problem in the presence of dynamic communication faults.

For those graphs where $con(G) < deg(G)$ there is a gap between possibility and impossibility. Closing this gap is clearly a goal of future research.

3 Broadcast with Mobile Omissions

Among the primitive operations of a distributed computing environment, a basic one is *broadcasting*: a single entity, the initiator, is in possession of some information, and the task is to inform all other entities. A generalization of this problem is the *wake up* in which one or more entities, the initiators, are *awake* while all others are *asleep*, and the task is to transform all entities into *awake*. Clearly, the broadcast is a wake-up with a single initiator; conversely, a wake-up is a broadcast when more than one entity initially has the information.

The solution to both problems is important because of the basic nature of these problems; it is also crucial, as we have seen, in the resolution of the agreement problem, and thus for the computability of non-trivial tasks. For these reasons, broadcast in the presence of mobile communication faults has been the object of extensive investigations, mostly in the case of **Omit**. Let us denote by $T_{\mathbf{type}}(G)$ the *minimum* time ensuring that broadcast is possible in G in spite of a number of mobile communication faults of *type* not exceeding the bounds discussed before.

3.1 General Bounds

Consider the case when the communication errors are just *omissions*. We have just seen that broadcasting is possible if and only if $F \leq con(G) - 1$: the initiator, starting at time $t = 0$

transmits the information to all its neighbours for $T(G)-1$ consecutive time units; an entity receiving the information for the first time at time $t < T(G)$ transmits the information to all its other neighbours at each time step until time $T(G)-1$. Let us call *BcastOmit* this protocol; then the *minimum* timeout value $T^*(G)$ ensuring that the *BcastOmit* is correct in G in spite of $con(G)-1$ omissions per clock cycle is precisely $T_{\mathbf{omit}}(G)$:

$$T_{\mathbf{omit}}(G) = T^*(G) \tag{1}$$

Clearly, $T_{\mathbf{omit}}(G) \leq T_{\mathbf{corr+omit}}(G)$ and $T_{\mathbf{omit}}(G) \leq T_{\mathbf{add+omit}}(G)$. On the other hand, By Theorems 7 and 8 we have $T_{\mathbf{corr+omit}}(G) \leq T^*(G)$ and $T_{\mathbf{add+omit}}(G) \leq T^*(G)$, respectively. Thus, by (1) we have

$$T_{\mathbf{corr+omit}}(G) = T_{\mathbf{add+omit}}(G) = T^*(G) \tag{2}$$

Hence the key point is to determine (an upper bound on) the value $T^*(G)$. It is not difficult to verify that

Lemma 3 $T^*(G) \leq (n-2)\ con(G)\ +1$.

Proof. In G there are at least $con(G)$ edge-disjoint paths between any two entities x and y; furthermore, each of these paths has length at most $n-1$. In *BcastOmit* , an initiator x sends a message along all these $con(G)$ paths. At any time instant, there are $F < con(G)$ omissions; this means that at least one of these paths is free of faults. That is, at any time unit, the message from x will move one step further towards y along one of them. Since these paths have length at most $n-1$, after at most $(n-2)con(G)\ +1 = n\ con(G)\ -2\ con(G)\ +1$ time units the message from x would reach y. ■

This upper bound is rather high, and depending on the graph G can be substantially reduced. Currently, the best available upper bounds are due to Chlebus, Diks, and Pelc [**?**]:

Theorem 11 [**?**] $T^*(G) = O(diam(G)^{con(G)})$.

Theorem 12 [**?**] $T^*(G) = O(con(G)^{diam(G)/2-1})$.

Which general bound is better (i.e., smaller), depends on the graph G. Interestingly, in a hypercube H, as well as in other specific topologies, all these estimates are far from accurate.

3.2 Specific Bounds

Consider a d-dimensional *hypercube* H; in H, between any two nodes x and y there are d edge-disjoint paths of length at most d. That is, the paths in the proof of lemma 3 have length at most $\log n$ (and not n), yielding the obvious $\log^2 n$ upper bound to broadcast time. This upper bound on $T^*(G)$ has been first decreased to $O(\log n)$ by Ostromsky and Nedev [**?**], then to $\log n + O(\log\log n)$ by Fraigniaud and Peyrat [**?**], to $\log n + 7$ by De Marco and Vaccaro [**?**] who proved also a *lower bound* $T^*(H) \geq \log n + 2$, until finally Dobrev and Vrťo [**?**] have shown that $\log n + 2$ clock cycles suffice ! In other words, with only two time units more than in the fault-free case, broadcast can tolerate up to $\log n - 1$ message losses per time unit. Summarizing,

Theorem 13 [?, ?] $T^*(H) = \log n + 2$.

The time to complete the broadcast in a d-dimensional hypercube has also been investigated when the number F of mobile faults is less than $d-1$; Dobrev and Vrťo [?] have shown that if $F \leq d-3$ then the minimum broadcast time is $\log n$, while if $F = d-2$ the broadcast time is always at most $\log n + 1$.

The case of d-dimensional general *tori* R has been investigated by Chlebus, Diks and Pelc [?] who showed that $T^*(R) = O(diam(R))$. In case of the d-dimensional k-ary even torus, De Marco and Rescigno [?] proved that $T^*(R) \leq diam(R) + 1$ provided k is limited by a polynomial in d. Without this assumption Dobrev and Vrťo [?] showed that broadcast would be completed in at most one more time unit. Summarizing

Theorem 14 [?] $T^*(R) \leq diam(R) + 2$.

The case of a *complete graph* K is particularly interesting. Indeed, it is the first graph in which the effects of mobile faults were investigated [?, ?]. The interest derives also from the fact that, while no non-trivial computation is possible with $F = con(K) = n-1$ omissions per clock cycle, if the mobile omissions per clock cycle are $n-2$ or fewer then any computable function can be computed in *constant* time. This is due to the fact that broadcast can be performed in four cycles:

Theorem 15 [?] $T^*(K) \leq 4$.

To see why this is true, consider the initiator x starting a broadcast at time $t = 0$; since at most $n-2$ messages may be lost, at time $t = 1$ at least one more entity has the information; this means that these two entities send a total of $2(n-1)$ messages of which at most one in each direction is between the two sending entities (and thus wasted), and at most $n-2$ are lost. This means that $2(n-1) - 2 - (n-2) = n-2$ messages go towards the other entities. Since these messages come from two different sources (x and y), at least $(n-2)/2 = n/2-1$ new entities will receive the information. That is, at time $t = 2$ at least $2+n/2-1 = n/2+1$ entities have the information and forward it to all their neighbours. In the next step at least $n-1$ entities have the information because $n/2+1$ omissions are required to not inform any new entity, and $\lfloor (n-2)/(n/2+1) \rfloor = 1$, ensuring that at time $t = 4$ the broadcast is completed. A more detailed analysis depending on the number of initiators in each round has been done by Liptak and Nickelsen [?].

To achieve this time, the protocol *BcastOmit* requires all entities receiving the information to forward the information to all neighbours at each time unit of its execution. Thus, the total number of messages is in the worst case $(n-1)(3n+1)$. If the concern is on the amount of communication instead of time, a better broadcast protocol has been designed by Dobrev [?] if the graph has weak sense of direction; in this case, for example, it is possible to broadcast in time $O(\log n)$ exchanging only $O(n \log n)$ messages.

4 Function Evaluation in Complete Networks

While in the previous sections we considered the problem of reaching unanimity and broadcasting in spite of faults, we now widen our view towards the computation of (Turing-computable) functions on the input.

4.1 Problem and Basic Strategies

The computation of (Turing-computable) functions on the input is a fundamental class of problems in distributed computing. The functions of interest include specific ones, such as counting the number of 1's among the input bits of the entities or reaching strong majority, as well as arbitrary, general ones. For initial value r_i of entity x_i, $i = 1, \ldots, n$, function computation requires that the result is known to all entities after a finite number of clock cycles, that is, the result value v_i must be the same, namely the function value, for all i.

We limit, however, the type of network to the complete graph on n entities. That is, we consider n entities distributed over a network, where each entity is directly connected to each other entity through a distinct one-way link (often called *point-to-point* model of communication). Each entity has a unique identity, e.g., one of the numbers 1 to n; that is, entities are not anonymous, and the size of the complete network is known to each entity. Now, the objective of a computation is to evaluate a non-constant function on all entities' inputs.

Obviously, arbitrary function computation is a stronger computational requirement than mere unanimity (or any other form of agreement): Whenever a distributed computing environment can compute a function, it can also agree unanimously on the function value. Therefore, the upper bounds on the number of tolerable faults for unanimity hold also for function computation. In particular, for omissions only, the upper bound of $F = con(G) - 1$ on the number of faults per clock cycle translates into $F = n - 2$, because there are $n - 1$ edge disjoint paths between any two entities in a complete graph.

We will consider two basic strategies of computing a function. In the first strategy, each entity is made to know enough information on input values to evaluate the desired function; in general, here each entity needs to know all input values. Then, each entity computes the function value locally.

In the second strategy, the function value is computed at some entity and made known to all others. Here we make use of the fact that for not too many communication faults, some entity will for sure know all the input values; then, the function value needs to be communicated to all entities. For both strategies, it is essential that we can propagate a value to all entities, i.e., broadcast a value.

All results of this section are from [**?**], unless otherwise specified.

4.2 Omissions

We have seen that for omissions only, broadcast works in 4 clock cycles when $F \leq n - 2$ (Theorem 15). The broadcast can be used to propagate all input values to all entities, where

each message contains both the identifier and the input value of an entity. Note that the input value is not limited to be a single bit, but may just as well be a bitstring of arbitrary length (when we will look at omissions and corruptions next, the situation will no longer be this simple). After all input values have reached all entities, the desired function value can be computed at each entity locally:

Theorem 16 *In* **Omit**, *an arbitrary function of the input can be computed in 4 clock cycles, in spite of $F = n - 2$ mobile communication faults per clock cycle.*

This result is tight, since with $n - 1$ or more faults, not even strong majority can be achieved (see Theorem 1).

The problem of coping with omissions becomes far more difficult when we consider arbitrary anonymous networks. Due to the anonymity, we cannot distinguish between inputs of different entities, and hence we are unable to propagate all input values to all entities and compute a function locally. Already the problem of determining the number of occurrences of all input values (their multiplicity) turns out to be very difficult and has been open for some time. A beautiful solution [**?**] shows, however, that it is solvable whenever the number of faults is smaller than the edge connectivity and the network size is known:

Theorem 17 [**?**] *In* **Omit** *in an anonymous network of known size and of edge connectivity $con(G)$, the multiplicity of the input values can be computed in spite of $F = con(G)-1$ mobile communication faults per clock cycle.*

As a consequence, any function on that multiplicity can also be computed, such as for instance the **XOR** of binary inputs.

4.3 Omissions and Corruptions

We show that the upper bound $n - 2$ on the number of tolerable omission and corruption faults for unanimity is tight for arbitrary function computation.

4.3.1 Basic Tools

Let us realize a communication of an arbitrary value by a series of communications of bits (we call this *bitwise communication*). Due to the corruptions, we cannot distinguish for a received bit value between a 0 and a 1 that has been sent. We choose to limit ourselves to send either nothing or a 0.

Whenever we only need to broadcast a specific input value that is fixed before the algorithm starts to execute, such as a 0 for the Boolean **AND** of input bits or a 1 for the **OR**, this specific value is represented by the 0 that entities send. This leads immediately to the following:

Lemma 4 *In* **Omit+Corr**, *an arbitrary fixed value can be broadcast in 4 clock cycles, in spite of $F = n - 2$ mobile communication faults per clock cycle.*

Proof. To broadcast value v, each entity holding a v sends a 0 in the first cycle; all others are silent, i.e., they send nothing. Each entity receiving a 0 or a 1 knows that the sent value represents v. In the following cycles, each entity knowing of a value v propagates a 0. In at most 4 cycles, all entities know of v, if there is an input value v in the system. ■

Along the same lines of thought, we can broadcast a bitstring bit by bit with the *time slice* technique that we presented earlier: A 0 being sent represents a 0 in even numbered cycles and a 1 in odd clock cycles. Then, a bit can be sent in each *communication cycle* consisting of an even and an odd cycle. Now, various possibilities exist for *bitstring broadcasting*.

Techniques *Bitstring Broadcasting*:

1. **Technique** *Bitstring Timeslicing*:

   The most immediate approach is to broadcast bits in time slices. Since time slicing requires two cycles (one communication cycle) per bit to be communicated, in 4 communication cycles a one bit message will have been broadcast to every entity. In other words, if an entity has not received a message in these 4 communication cycles, nothing has been sent. We can use the freedom in the encoding of one message bit in two binary symbols to cope with the problem of terminating the bitstring, by encoding the bits as follows. In a communication cycle,

   - $0, 0$ stands for 0, and the message continues;
   - $\Omega, 0$ stands for 1, and the message continues;
   - $0, \Omega$ stands for 0, and the message ends;
   - Ω, Ω stands for 1, and the message ends.

   For a message of size s (measured as its length in bits), in at most $4s$ communication cycles, i.e., in $8s$ clock cycles, the bitstring reaches all entities.

2. **Technique** *Length Timeslicing*:

   Instead of sacrificing an extra bit for each bit of the entire message, we might as well first send the length of the message (with time slicing), and then send the message by bitwise broadcast. In this case, each bit of the message can be encoded in one binary symbol of $\{0, \Omega\}$ to be broadcast. For a message of size s, we need to broadcast $\lceil \log s \rceil$ bits with time slicing, plus an additional s bits without time slicing, in $8\lceil \log s \rceil + 4s$ clock cycles altogether.

   Alternatively, the folklore concept of representing a bitstring in a *self-delimiting* way (see the section on binary strings in [**?**]), applied to our situation, would first send the length s of the bitstring in unary, by sending a 0 in each of s clock cycles, followed by one clock cycle with no communication to indicate that the unary sequence ends, followed by the bitwise broadcast of the message in the next s clock cycles. This would take $4(2s + 1) = 8s + 4$ clock cycles altogether and therefore not be attractive in our setting.

3. **Technique** *Iterated Length Timeslicing*:

   The process of first sending the length of a message by time slicing, and then sending the message itself without time slicing may be applied repeatedly, in a fashion minimizing the number of clock cycles needed, if desired.

We summarize this discussion as follows:

Lemma 5 *In* **Omit+Corr**, *a string of size s can be broadcast in $8s$ clock cycles, in spite of $F = n - 2$ mobile communication faults per clock cycle. It can alternatively be broadcast in $8\lceil \log s \rceil + 4s$ clock cycles.*

4.3.2 Input Bits

For the special case of input bits only (instead of input strings), Lemma 4 implies that the Boolean **AND** and **OR** can both be computed in 4 clock cycles, in spite of $n - 2$ faults. At the same time with the same number of faults, it can also be determined whether the number of 1's in the input is at least k, for a fixed value k. To see whether at least k 1's occur in the input, each entity holding a 1 sends it in the first clock cycle; all other entities are silent. At least two entities will have received a bit (perhaps corrupted) for each of the 1's that occur in the input of other entities; we say such an entity knows all the 1's. However, no entity can detect at this stage whether it knows all the 1's. In the second cycle, each entity knowing of at least k 1's (including its own input value) starts broadcasting a 1 through the network. That is, the fixed value 1 is broadcast as outlined in the proof of Lemma 4. Since at least two entities start this broadcast in the second cycle, in at most 4 cycles in total each entity knows the result: if a 1 has reached the entity, the answer is affirmative. Similarly, one can solve the other three variants of the problem that ask for at least k 0's, at most k v's, and at most k 0's (note that with a binary alphabet, at least k 1's is equivalent to at most $n - k$ 0's). We summarize as follows:

Theorem 18 *In* **Omit+Corr** *with bits as input, it can be determined in 4 clock cycles whether there are at least (at most) k 0's (1's) in the input, in spite of $F = n - 2$ mobile communication faults per clock cycle.*

Functions on the multiset of inputs

We start to consider more general function computation by looking at functions on the multiset of inputs; after that, we will consider functions on the vector of inputs. For functions on the multiset of input bits, it is enough that each entity knows the exact number of 0's and 1's in the input. This can be determined by a binary search over the integers from 1 to n for the precise number of 1's in the input. To this end, each processor holding an input 1 sends it in the first cycle. Then, the three additional cycles to decide whether there are at least k 1's in the input are repeated exactly $\lceil \log n \rceil$ times, for values of k defined by binary search. At the end, each processor knows the number of 1's and 0's in the input, and can compute the function. Hence, we have the following:

Theorem 19 *In* **Omit+Corr** *with bits as input, any function on the multiset of inputs (as opposed to the input vector) can be determined in* $1 + 3\lceil \log n \rceil$ *clock cycles, in spite of* $F = n - 2$ *mobile communication faults per clock cycle.*

Functions of the input vector
Let us now turn to the computation of a function of the input vector. In the situation in which a function on a multiset of the inputs can be computed, as described above, all entities know the number of 1's in the input, say k_1, after $1 + 3\lceil \log n \rceil$ cycles. Now, each entity can check whether it has known of k_1 1's already after the first cycle (again, including its own input value). If so, an entity now also knows the input vector, and there are at least two of these entities in the system. These entities now compute an arbitrary function f on the input vector I, and then broadcast the result $f(I)$ by bitstring broadcast. Naturally, only entities having computed $f(I)$ will initiate the bitstring broadcasting; all other entities will only join in the broadcast.

Theorem 20 *In* **Omit+Corr** *with bits as input, any function f on the vector I of input bits can be computed in* $1 + 3\lceil \log n \rceil + 8\lceil \log |f(I)| \rceil + 4|f(I)|$ *clock cycles, where* $|f(I)|$ *is the length of the representation of f(I), in spite of* $F = n - 2$ *mobile communication faults per clock cycle.*

4.3.3 Input Strings

For bitstrings as inputs, we still stick to communicating messages bitwise, due to possible corruption. To compute an arbitrary function, we collect all input strings at all entities, and evaluate the function locally at each entity.

First, we perform a *length determination* step: If not known already, we first guess the length s of a longest input string by means of unbounded search for s. Recall that unbounded search goes through powers of two, starting at 2, until $s \leq 2^i$ for the first time for some i. It then performs binary search for s between 2^{i-1} and 2^i. A single one of these probes with value v in unbounded search works as follows. Each entity holding an input of length at least v initiates the broadcast of a message. In 4 cycles, the outcome of the probe is known to each entity: If no communication bit has been received in the last 4 cycles, none of the other entities holds an input of length at least the probe value v, and otherwise at least one other entity has such an input. Since unbounded search terminates after at most $2\lceil \log s \rceil$ probes, $8\lceil \log s \rceil$ cycles suffice.

Then, in a *length adjustment* or *padding* step, all inputs are adjusted to maximum length locally at each entity. We assume this can be done, e.g., by adding leading zeroes if the inputs represent binary numbers. (Otherwise, we apply the technique of doubling the length of each input first and marking its end, by applying the scheme described in the Bitstring Timeslicing Technique, and then we can pad the bitstring as much as needed beyond its designated end; this leads to an extra factor of 2 inside the time bound that we do not account for below.)

Finally, each entity in turn, as determined by a global order on entity identities, communicates its input string bit by bit according to the bitstring broadcasting scheme. This

consumes no more than $8s$ cycles for each of the n inputs. Now, each entity knows all input strings and computes the function value locally. In total, this adds up as follows:

Theorem 21 *In* **Omit+Corr** *with bitstrings as inputs, any function can be computed in at most* $8\lceil \log s \rceil + 8ns$ *clock cycles, where* s *is the length of a longest input bitstring, in spite of* $F = n - 2$ *mobile communication faults per clock cycle.*

Specific functions, again, can be computed faster. For instance, the maximum (minimum) value in a set of input values represented as binary numbers can be found by unbounded search for that number. We get:

Theorem 22 *In* **Omit+Corr** *with bitstrings as input, the maximum (minimum) value in the input can be computed in at most* $8\lceil \log v \rceil$ *clock cycles, where* v *is the result value, in spite of* $F = n - 2$ *mobile communication faults per clock cycle.*

Similarly, the rank r of a given query value v in the set of binary input numbers (that is, the relative position that v would have in the sorted input sequence) can be determined by having each entity whose input value is at most v communicate in the first cycle; all other entities are silent. In the second cycle, each entity computes the number n_i of messages it has received, and increments it if its own input value is at most v. The largest value of any n_i now equals r and is known to at least two entities. Since $1 \leq n_i \leq n$ for all i, r can be determined by binary search in at most $4\lceil \log n \rceil$ clock cycles in total. Therefore, we get:

Theorem 23 *In* **Omit+Corr** *with bitstrings as input, the rank among the input values of a given value can be computed in at most* $4\lceil \log n \rceil$ *clock cycles, in spite of* $F = n - 2$ *mobile communication faults per clock cycle.*

By using similar techniques, we can select the k-th largest input value, compute the sum of the input values (interpreted as binary numbers), and compute various statistical functions on the inputs, like mean, standard deviation, variance, within similar time bounds, i.e., faster than arbitrary functions.

4.4 Applications of the Basic Techniques

4.4.1 Corruptions

We have seen that for the problem of reaching unanimous agreement, an arbitrary number of corruptions can be tolerated. The same is true for the computation of arbitrary functions, essentially for the same reason. Bits can be communicated without ambiguity, e.g., by representing a 0 by the communication of a bit, and a 1 by the absence of a communication.

To compute a function, first all inputs are collected at all entities, and then the function is evaluated locally. The collection of all input strings at all entities starts by a *length determination* step for the length s of the longest input string by unbounded search in $2\lceil \log s \rceil$ probes, where each probe takes only one clock cycle: For probe value j, an entity

sends a bit if its own input value is at least j bits long, and is otherwise silent. Hence, the length determination takes $2\lceil \log s \rceil$ clock cycles. Then, we perform a *padding* step locally at each entity so as to make all strings equally long. Finally, bit by bit, all input values are sent simultaneously, with the unambiguous encoding described above.

Because the communication of all inputs to all entities takes only s clock cycles, we get:

Theorem 24 *In* **Corr**, *an arbitrary function of the input can be computed in* $2\lceil \log s \rceil + s$ *clock cycles, in spite of* $F = n(n-1)$ *mobile communication faults per clock cycle.*

Specific functions can, again, be computed faster in many cases. In one such case, a function asks for the number of input strings that have a certain, fixed property. Locally, each entity determines whether its input has the property, and sends a bit unambiguously telling the local result to all other entities. We therefore have:

Theorem 25 *In* **Corr**, *any function of the form* $f(I) = |\{j \in I : P(j) \text{ holds } \}|$ *for a computable predicate* P *on the input* I *can be computed in* 1 *clock cycle, in spite of* $F = n(n-1)$ *mobile communication faults per clock cycle.*

4.4.2 Additions

If only spontaneous additions can occur, but no corruptions or loss of messages, all links may be faulty at all times, with no negative effect, because each entity simply sends its input string without ambiguity, and then each entity computes the desired function locally:

Theorem 26 *In* **Add**, *any function can be computed in* 1 *clock cycle, in spite of* $F = n(n-1)$ *mobile communication faults per clock cycle.*

4.4.3 Additions and Corruptions

For the communication of single bits, additions and corruptions are isomorphic to omissions and corruptions, in the following sense. For omissions and corruptions, we made use of an unambiguous encoding that, e.g., communicates a 0 to indicate a 0, and that sends nothing to indicate a 1. With an adapted behavior of the entities, this very encoding can also be used with additions and corruptions: Just like omissions were overcome by propagating a bit for 4 cycles, additions are overcome by propagating the absence of a message for 4 cycles. More precisely, we associate a phase of 4 cycles to the communication of 1's. If an entity has input bit 1, it is silent in the first cycle of the phase. If an entity does not receive $n-1$ bits (one from each other entity) in a cycle, it knows that a 1 is being communicated, and it is silent in all future cycles of this phase, thus propagating the 1. This isomorphism can be used in the computations of arbitrary functions on input strings and input bits. We therefore get:

Theorem 27 *In* **Add+Corr**, *arbitrary function computation tolerates the same number of mobile communication faults per clock cycle as in* **Omit+Corr**.

4.4.4 Additions and Omissions

For omissions only, our algorithms always communicate, for arbitrary function computation as well as for strong majority. Therefore, additions cannot occur, and hence, additions and omissions are not more difficult than omissions alone:

Theorem 28 *In* **Add+Omit**, *arbitrary function computation and strong majority tolerate the same number of mobile communication faults per clock cycle as in* **Omit**.

4.4.5 The Byzantine Case: Additions, Omissions, and Corruptions

Recall the *Acceptance Rule* that was used for unanimity in the Byzantine case (see Theorem 10), based on an application of *Reliable Neighbour Transmission*: in a communication of one bit from an entity x to a direct neighbour y of x, y selects the majority bit value that it has received from x on the $con(G)$ chosen paths within $t > (con(G)-1)(l-1)$ clock cycles, where l is the length of a longest one of all such paths. This rule guarantees correct communication of a bit from x to y if the number of Byzantine faults per clock cycle is limited to not more than $\lceil con(G)/2 \rceil - 1$. Applied to the Byzantine case in the complete network, this means that we can safely transmit a bit between any two entities in $n-1$ clock cycles, if there are not more than $\lceil (n-1)/2 \rceil - 1 = \lfloor (n-1)/2 \rfloor$ faults per cycle.

We can now build on the safe bitwise transmission by collecting all inputs of all entities at one distinguished entity, by then computing the function value there locally, and by then broadcasting the result to all other entities. To send one input bitstring between two entities, based on safe bitwise transmission, we can use the technique of doubling the length of the bitstring as in *Bitstring Timeslicing* to indicate its end, or the *Length Timeslicing* technique, or the *Iterated Length Timeslicing* technique. We therefore get:

Lemma 6 *In* **Omit+Add+Corr**, *a string of size s can be communicated between two entities in a complete network in $2s(n-1)$ clock cycles, in spite of $\lfloor (n-1)/2 \rfloor$ faults per clock cycle.*

Even if we do not care to overlap the bitstring communications in time, but simply collect and then distribute values sequentially, in $n-1$ rounds of collecting and distributing, the Byzantine function evaluation is completed:

Theorem 29 *In* **Omit+Add+Corr**, *any function can be computed in $2S(n-1)$ clock cycles, where S is the total length of all input strings and the result string, in spite of $\lfloor (n-1)/2 \rfloor$ faults per clock cycle.*

5 Other Approaches

The solutions we have seen so far are F-tolerant; that is, they tolerate up to F communication faults per time unit, where the value of F depends on the type of mobile faults in the system.

The solution algorithms, in some cases, require for their correct functioning the transmission of a large number of messages (e.g., protocol *BcastOmit*). However, in some systems, as the number of message transmissions increases, so does the number of lost messages; hence those solutions would not behave well in those faulty environments.

5.1 Probabilistic

An approach that takes into account the interplay between the amount of transmissions and the number of losses is the *probabilistic* one. It does not assume an a priori upper bound F on the total number of dynamic faults per time unit; rather, it assumes that each communication has a known probability $p < 1$ to fail. The investigations using this approach have focused on designing broadcasting algorithms with low time complexity and high probability of success, and have been carried out by Berman, Diks, and Pelc [**?**] and by Pelc and Peleg [**?**]. The drawback of this approach is that obviously the solutions derived have no deterministic guarantee of correctness.

5.2 Fractional

The desire of a deterministic setting that explicitly takes into account the interaction between number of omissions and number of messages has been the motivation behind the *fractional* approach proposed by Královič, Královič and Ruzicka [**?**]. In this approach, the amount of dynamic faults that can occur at time t is not fixed but rather a linear fraction $\alpha\ m_t$ of the total number m_t of messages sent at time t, where $0 \leq \alpha < 1$ is a (known) constant. The advantage of the fractional approach is that solutions designed for it tolerate the loss of up to a fraction of all transmitted messages [**?**]. The anomaly of the fractional approach is that, in this setting, transmitting a single message per communication round ensures its delivery; thus, the model leads to very counterintuitive algorithms which might not behave well in real faulty environments.

To obviate this drawback, the fractional approach has been recently modified by Dobrev, Královič, Královič and Santoro [**?**] who introduced the *threshold fractional* approach. Like the fractional approach, the maximum number of faults occurring at time step t is a function of the amount m_t of messages sent in step t; the function is however

$$F(m_t) = \max\{T_h - 1, \lfloor \alpha\ m_t \rfloor\}$$

where $T_h \leq c(G)$ is a constant at most equal to the connectivity of the graph, and α is a constant $0 \leq \alpha < 1$. Note that both the traditional and the fractional approaches are particular, extreme instances of this model. In fact, $\alpha = 0$ yields the traditional setting: at most $T_h - 1$ faults occur at each time step. On the other hand, the case $T_h = 1$ results in the *fractional* setting. In between, it defines a spectrum of new settings never explored before.

The only new setting explored so far is the *simple threshold*, where to be guaranteed that at least one message is delivered in a time step, the total amount of transmitted messages in that time step must be above the threshold T_h; surprisingly, broadcast can be completed in (low) polynomial time for several networks including rings (with or without knowledge of n),

complete graphs (with or without chordal sense of direction), hypercubes (with or without orientation), and constant-degree networks (with or without full topological knowledge) [**?**].

6 Conclusions

For graphs whose connectivity is the same as the degree, there exists a precise map of safe and unsafe computations in the presence of dynamic faults, generalizing the existing results for complete graphs. For those graphs where $con(G) < d(G)$, the existing results leave a gap between possibility and impossibility. Closing this gap is the goal of future investigations. Preliminary results indicate that neither parameter provides the "true bound" for arbitrary graphs: there are graphs where $d(G)$ is too large a parameter and there are networks where $con(G)$ is too low. An intriguing open question is whether there actually exists a single parameter for all graphs.

When the faults include additions and corruptions (with or without omissions) the current unanimity protocol assumes knowledge of the network. Is it possible to achieve unanimity without such a requirement?

With a few exceptions, the performance in terms of messages of the protocols have not been the main concern of the investigations. Designing more efficient reliable protocols would be of considerable interest, both practically and theoretically.

In the threshold fractional approach, the only new setting explored so far is the simple threshold. All the others are still unknown.