Fast Adaptable Uniform Consensus Using Global State Digests

Andrey Brito[1], **Francisco Brasileiro**[1,2], **Walfredo Cirne**[1,2]
Universidade Federal de Campina Grande
[1]Coordenação de Pós-Graduação em Informática
[2]Coordenação de Pós-Graduação em Engenharia Elétrica
Av. Aprígio Veloso, 882 - 58.109-970, Campina Grande, PB, Brazil
Phone: +55 83 310 1433 Fax: +55 83 310 1365
{fubica,andrey,walfredo}@dsc.ufcg.edu.br

Abstract

Protocols that solve the consensus problem have been widely recognized as important building blocks for the design of reliable distributed systems. This fact explains why considerable amount of work has been devoted both to establish the minimal system requirements that allow a solution to the problem, as well as to provide efficient protocols to solve it. We propose the use of global state digests *to design efficient and adaptable consensus protocols. A global state digest is a bounded and consistent summarized representation of the local states of all processes that run the protocol. By frequently providing processes with new global state digests, it is possible to allow processes to terminate the protocol soon after the minimal condition necessary to solve the problem holds, whatever are the contention levels experienced by the system. We present the design of a family of fast adaptable consensus protocols using this abstraction. Further, a global state digest provider can be implemented whenever the same assumptions required to implement perfect failure detectors hold (basically the ability to convey a bounded amount of information within a bounded interval of time).*

Keywords*: agreement protocols; perfect failure detection; consistent global state; synchronous and asynchronous distributed systems; wormholes.*

1 Introduction

In this paper we focus on the fundamental problem of reaching consensus among a set of n processes that communicate exclusively via the exchange of messages [11]. In the consensus problem, each process p_i proposes a value v_i and every correct process must decide for the same common value v among the values proposed, despite the possible crashes of up to f processes, $f < n$. Designing a protocol that guarantees these properties, however, is not trivial. There are two main sources of difficulties associ-

ated with the design of a fault-tolerant distributed consensus protocol, namely: i) the lack of synchrony guarantees provided by the underlying distributed system; and ii) the occurrence of failures in both processing and communication.

Synchronous systems provide strong synchrony guarantees for both processing schedule and communication delays. Because of this, detection of benign failures (such as crashes) within synchronous systems is straightforward. Therefore, solutions to the consensus problem for this kind of system have long been known [7]. On the downside, the strong synchrony guarantees provided by synchronous systems require an a priori knowledge of the worst case workloads to which the system will ever be subject. This makes it more difficult to develop such systems and reduces the applicability of fault-tolerant distributed protocols designed to execute over a synchronous system.

On the other hand, in a pure asynchronous system model, no synchrony guarantees are given (in fact the very notion of time is absent), thus, any distributed system can be considered a pure asynchronous system. Unfortunately, it is well known that most practical fault-tolerant distributed problems are impossible to be solved in this kind of system. Particularly, it has been proved that there is no deterministic solution to the consensus problem in pure asynchronous systems subject to even a single crash failure [12].

This result has prompted researchers to seek for the minimal synchrony guarantees that a distributed system must provide in order to allow fault-tolerant solutions to fundamental distributed problems, such as consensus. This effort is backed up by the observation that, although most off-the-shelf distributed systems are not synchronous, they do have some level of synchrony, and are, therefore, generally classified as partially synchronous systems [8, 10].

Within this context, the abstraction of unreliable failure detectors [5] is an important contribution. An unreliable failure detector is a "black-box" that is associated with each process of an asynchronous distributed system and encapsulates the synchrony required to solve fault-tolerant distributed problems. This synchrony is specified via a pair of properties that describe the behavior of classes of failure detectors. They define the failures that the failure detectors are able to detect (their *completeness* property) and the mistakes that they are allowed to make (their *accuracy* property). It has been shown that the weakest failure detector class that allows a deterministic solution to consensus, named $\diamond S$, must provide the following properties: eventually every process that crashes is permanently suspected by every correct process (*strong completeness*); and, there is a time after which some correct process is never suspected by any correct process (*eventual weak accuracy*) [6].

Following this result, a number of consensus protocols based on failure detectors of the class $\diamond S$ have been proposed (*e.g.* [16, 14, 13, 9, 15]). They all share the following characteristics: i) they are only able to tolerate a minority of faulty processes, *i.e.* they require $f < n/2$; and, ii) they are *indulgent* [9] in relation to the failure detector, *i.e.* even if the failure detector misbehaves and does not provide any synchrony guarantees, the safety properties of consensus are preserved. Other solutions to the consensus problem based on stronger failure detectors have also been proposed [5, 1]. They are not indulgent towards the failure detector, but allow greater resilience to faults (typically, they are able to tolerate up to $n - 1$ faults).

Implementations of stronger failure detectors require more synchrony guarantees from the underlying distributed system, which, as mentioned before, are more difficult to be found in most systems available

today. To overcome this problem, hybrid architectures (*wormholes*) may be used [18, 17]. The idea is to design protocols that assume an asynchronous system model and make no explicit requirement for synchrony guarantees, relying only on a strong failure detector (for instance, a perfect one) that encapsulates the required synchrony. In order to implement a perfect failure detector, the asynchronous system is augmented with a synchronous subsystem that is solely used to support the implementation of the failure detector. Note that the synchronous subsystem corresponds to just a small and well defined portion of the whole system, thus, rendering the difficult task of engineering such subsystem much simpler.

In a previous work [3] we discussed the implementation of a hybrid system, which relied on a hardware-based synchronous subsystem. The usage of this system was exemplified with a service, the Global State Digest Provider service (GSDP), that could be used to solve consensus. Later, we showed how a cheap and safe synchronous subsystem could be built based on COTS components [4]. In this paper, we give a detailed formalization of the GSDP abstraction. We then use the abstraction to design a consensus protocol, providing detailed description of the protocol as well as correctness proofs and a performance evaluation.

The rest of the paper is structured in the following way. Section 2 describes the system model assumed, gives a general description of the abstraction of a global state digest provider, and formally defines the uniform consensus problem. In Section 3, we present a family of uniform consensus protocols based on a particular instantiation of the global state digest provider abstraction (the proof of correctness of the protocols proposed is presented in Appendix A). In Section 4 we evaluate the performance of the proposed protocols and compare their performance with that of other protocols previously published. Section 5 concludes the paper with our final remarks.

2 System Model and Definitions

System Model The system model is patterned after the one described in [12]. It consists of a finite set Π of n processes, $n > 1$, namely, $\Pi = \{p_1, \ldots, p_n\}$. A process can fail by *crashing*, *i.e.* by prematurely halting, and a crashed process does not recover. A process behaves correctly (*i.e.* according to its specification) until it (possibly) crashes. At most f processes, $f < n$, may crash.

Processes are completely connected via a reliable network. Processes communicate with each other by message passing through the communication channels that comprise the reliable network: there are no message creation, alteration, duplication or loss. Further, there are assumptions neither on the relative speed of processes nor on message transfer delays.

The Global State Digest Provider To circumvent the impossibility result of [12] and allow the implementation of fast and adaptable consensus protocols, we consider that the system is augmented with a synchronous subsystem that implements the abstraction of a *global state digest provider* (GSDP). A *global state digest* (GSD) is a protocol-specific summarized description of the relevant events that happened within the system during a particular time interval, including an indication of the processes that have crashed.

Each process has access to the GSDP via its local GSDP module. This module is able to perform

some bounded computation in a timely way and to exchange, through a reliable network, a bounded number of messages whose sizes are also bounded. These restrictions allow end-to-end communication delays between any two correct GSDP modules to be bounded by a known constant δ. We assume that each GSDP module has access to a local clock that measures a clock interval time Δt in a real time interval $\Delta t'$, $\Delta t \times (1 - \rho) \leq \Delta t' \leq \Delta t \times (1 + \rho)$, where ρ is a known constant. Moreover, we assume that a process and its associated GSDP module form a single unit, thus a crashed (resp. correct) process also implies a crashed (resp. correct) GSDP module. This assumption is supported by the following arguments: (1) it is possible to implement the system such that both the GSDP and the process execute in the same node [4], thus, if the node crashes, the GSDP and the process will both fail; (2) if the GSDP could fail alone, the process remains stopped since it depends on the GSDP to make progress; also, it would not be considered in any distributed computation since the GSDP would not propagate its information; finally, (3) if the process fails the operating system will break the link between the process and the GSDP (for example, this link could be through an opened special file), and the GSDP will stop, propagating the failure information.

The Uniform Consensus Problem Before presenting protocols to solve the uniform consensus problem, let us recall the formal definition of the problem. In the uniform consensus problem, every process p_i *proposes* a value v_i and all correct processes have to *decide* on some value v, in relation to the set of proposed values. More formally, the uniform consensus problem is defined by the following properties [12]:

- **termination**: every correct process eventually decides some value;
- **uniform integrity**: every process decides at most once;
- **uniform validity**: if a process decides for the value v, then v was proposed by some process; and
- **uniform agreement**: no two processes decide differently.

3 Designing Uniform Consensus Protocols Supported by a Global State Digest Provider

We start the presentation of the protocols by defining the data structure of the GSDs that are delivered by the GSDP. For the consensus[1] protocol presented in this paper, a suitable GSD contains the following fields:

- *detection_vector*: a status vector with n bits, in which element i represents the operational status of process p_i (it is initially set to 0, and it is set to 1 if p_i is faulty);
- *reception_matrix*: a $n \times n$ matrix of bits in which the bit $[i, j]$ indicates whether p_i has received a message from p_j or not (it is initially set to 0, and it is set to 1 if the message has been received); and
- *consensual_identity*: an identity field with $\lceil log_2 n \rceil$ bits used to hold the identity of the process that provides the consensual value for the protocol (it is initially set to $\perp$).

[1] For the sake of brevity, from now on we will use simply the term consensus when referring to uniform consensus.

The family of consensus protocols supported by a GSDP have a synchronous part and an asynchronous part. There are two parameters that differentiate each member of the family of protocols. The first one, named *quorum*, defines the maximum number of processes that are necessary to "elect" the process that provides the consensual value. This parameter affects only the synchronous part of the protocol. The value of *quorum* is such that $f + 1 \leq quorum \leq n$. The second parameter, named *proposers*, defines the number of processes that are allowed to propose a value in the execution of the protocol. It affects only the asynchronous part of the protocol, and its value is such that $f + 1 \leq proposers \leq n$. A pair of values for the parameters *quorum* and *proposers* defines a particular protocol. Thus, each member of the family of protocols is denoted by *GSDP-consensus(q,p)*, where q and p are the *quorum* and *proposers* parameters, respectively. In the next two subsections we describe the functioning of the synchronous and asynchronous parts of the protocols.

3.1 The Synchronous Part of the Protocol GSDP-consensus(q, p)

The synchronous part of the protocol is executed by the GSDP modules that are associated with each process executing the consensus protocol. They exchange digests with the relevant events locally perceived in order to form *consistent* GSDs. More formally, a GSDP provides the following properties:

- **strong completeness of detection**: if some process p_j crashes, then eventually all GSDs delivered by every GSDP module have $detection\_vector[j] = 1$;
- **strong accuracy of detection**: if any GSD delivered has $detection\_vector[j] = 1$, then p_j has indeed crashed;
- **strong completeness of reception**: if some correct process p_i receives a message from some process p_j, then eventually all GSDs delivered by every GSDP module have $reception\_matrix[i, j] = 1$;
- **strong accuracy of reception**: if any GSD delivered has $reception\_matrix[i, j] = 1$, then p_i has indeed received a message from p_j;
- **validity**: if a GSD g has $g.consensual\_identity = x$ and f_{actual} is the number of entries in $g.detection\_vector$ set to 1 (*i.e.* the number of processes whose failures have been detected), then, there are at least $q - f_{actual}$ occurrences of i, $1 \leq i \leq n$, that satisfy: $g.detection\_vector[i] = 0 \wedge g.reception\_matrix[i, x] = 1$; also, if g has $g.consensual\_identity = \perp$, then, there is no p_x such that, there are at least $q - f_{actual}$ occurrences of i, $1 \leq i \leq n$, that satisfy: $g.detection\_vector[i] = 0 \wedge g.reception\_matrix[i, x] = 1$; and
- **write-once**: if a GSD is ever delivered with $consensual\_identity = x \neq \perp$, then every GSD delivered by any GSDP module that has $consensual\_identity \neq \perp$ also has $consensual\_identity = x$.

Each GSDP module $gsdp_i$ maintains two GSDs variables, named $ready\_for\_delivery_i$ and $local_i$. Initially, they have all bits of their $detection\_vector$ and $reception\_matrix$ set to 0, and their $consensual\_identity$ fields are set to $\perp$. To access its local GSDP module, a process p_i invokes the *getGSD* primitive. When p_i invokes this primitive, it receives in return the contents of the $ready\_for\_delivery_i$ variable. The $local_i$ variable is used to locally compute valid GSDs. Whenever a valid GSD is formed by a GSDP module $gsdp_i$, it copies $local_i$ into $ready\_for\_delivery_i$, thus making this newly formed GSD available for delivery.

To guarantee the *strong completeness of detection* and the *strong accuracy of detection* properties, the GSDP modules implement a perfect failure detector over the synchronous subsystem. The simplest way to implement such a failure detector in a synchronous system is to have every module broadcasting empty *heartbeat* messages at some a priori agreed rate. Assume that every correct GSDP module broadcasts a heartbeat message every τ units of time. Let $d = (\tau + \delta)(1 + \rho)$ and t_0 be a time after which all processes have already started the execution of the protocol. Thus, the failure of any GSDP module that happens at some time t, $t \geq t_0$, is detected by every correct module at latest by $t + d$. If a GSDP module $gsdp_i$ detects the failure of some process p_j, then it updates its $local_i$ GSD, such that $local_i.detection\_vector[j] = 1$. This guarantees the *strong completeness of detection* property, provided that $local_i$ eventually gets valid and is, therefore, copied into $ready\_for\_delivery_i$. Further, the upper bound on end-to-end communication delays of the synchronous subsystem (δ), together with the bound on the maximum drift rate (ρ), guarantee that detection is perfect, hence the *strong accuracy of detection* property also holds.

The *strong completeness of reception* and the *strong accuracy of reception* properties are also trivially met. Whenever a process p_i receives a protocol message from a process p_j, its associated GSDP module $gsdp_i$ is notified. Then, $gsdp_i$ sets $local_i.reception\_matrix[i, j] = 1$ and broadcasts a $\lceil log_2 n \rceil$-bit long message containing the index j of the element that has been set on its $local_i.reception\_matrix$ to all other GSDP modules. When a GSDP module $gsdp_k$ receives such a message it sets the corresponding element of its $local_k.reception\_matrix$. This guarantees *strong accuracy of reception*, provided that the local GSDs eventually get valid and are, therefore, copied into the respective $ready\_for\_delivery$ GSDs. Further, since the messages broadcast by a correct GSDP module are always received by all correct GSDP modules, *strong completeness of reception* is also guaranteed.

The most challenging aspect of the implementation of the GSDP is to guarantee that the *validity* and *write-once* properties are simultaneously met. The *validity* property is easy to be met, since it depends only on the local information gathered by a GSDP module. The difficulty arises when, after updating either the $detection\_vector$ or the $reception\_matrix$ of its $local$ GSD, a GSDP module $gsdp_i$ realizes that it must update $local_i.consensual\_identity$ (otherwise, no more valid GSDs may be made available). To preserve *validity*, from this point onwards, $local_i$ can only be copied into $ready\_for\_delivery_i$ if $local_i.consensual\_identity \neq \perp$. However, since it is possible that a GSDP module $gsdp_j$ fails while sending its heartbeat messages or while sending information about its $reception\_matrix$, some GSDP modules may receive such messages, while others do not. Thus, different GSDP modules might have different views on the value that should be used to set the $consensual\_identity$ field. Therefore, to guarantee the *write-once* property, they must elect a common process. There are several possibilities to achieve this, the simplest one being for a GSDP to atomically broadcast a message with its local view of which process should be elected. The first message that is atomically delivered defines the identity of the process to be used by all GSDP modules. The atomic broadcast protocol guarantees that the messages that are delivered by some GSDP module are delivered by every correct GSDP module; further, any two messages that are delivered by some GSDP module are delivered in the same order by all GSDP modules that deliver them [7]. This guarantees that all modules update their local GSD with the same value. Once the value of the $consensual\_identity$ of every correct GSDP module is set to the same value x, $x \neq \perp$, this value will never change, thus guaranteeing that the *write-once* property of the GSDP is met.

Algorithm 1 The pseudo-code of $gsdp_i$ associated with process p_i that executes GSDP-consensus(q, p)

```
% shared variables and function
local_i.detection_vector = ready_for_delivery_i.detection_vector = 0 % set all bits to 0
local_i.reception_matrix = ready_for_delivery_i.reception_matrix = 0 % set all bits to 0
local_i.consensual_identification = ready_for_delivery_i.consensual_identification = ⊥

while true do
   sleep for τ units of time
   send gsdp_i's heartbeat to all GSDP modules
end while
||
while true do
   when p_j's failure has been detected
      local_i.detection_vector[j] = 1
      if local_i satisfies the validity property then ready_for_delivery_i = local_i end if
   end when
end while
||
while true do
   when p_i notifies that p_j's consensus protocol message has been received
      local_i.reception_matrix[i, j] = 1
      send index j to all GSDP modules
      if local_i satisfies the validity property then ready_for_delivery_i = local_i end if
   end when
end while
||
while true do
   when index k has been received from gsdp_j
      local_i.reception_matrix[j, k] = 1
      if local_i satisfies the validity property then ready_for_delivery_i = local_i end if
   end when
end while
||
decision_reached = false
while not decision_reached do
   when local_i does not satisfy the validity property
      atomically broadcast the identity x of a process p_x whose message has been received by all correct processes
      waituntil atomically deliver some identity y
      local_i.consensual_identity = y; ready_for_delivery_i = local_i; decision_reached = true
   end when
end while
```

Remark. It has been shown in [5] that a solution to atomic broadcast is also a solution to consensus. Thus, a natural question is: why not use this atomic broadcast service to solve consensus for the application? We recall that the bandwidth of the synchronous subsystem must be judiciously used, and since the upper bound on the size of the messages that the applications will ever generate is unknown, they cannot directly use the synchronous subsystem.

Algorithm 1 gives the pseudo-code of the concurrent tasks that implement the GSDP module associated with a process p_i that executes GSDP-consensus(q, p).

3.2 The Asynchronous Part of the Protocol GSDP-consensus(q, p)

The asynchronous part of the protocol is structured as three concurrent tasks. In the first task (the *proposition* task) p processes send messages to the other processes containing their proposition values. The second task (the *listening* task) is responsible for receiving and storing the proposition messages that have been sent by the other processes. It also notifies its associated GSDP module of the proposition

messages that have been received. The final task (the *decision* task) is responsible for detecting that a decision can be made and that the execution of the protocol can be terminated. The decision task is also very simple. It enters a loop constantly querying its local GSDP module. Whenever a GSD is delivered such that the *consensual_identity* field is equal to some x, $x \neq \perp$, it verifies if it has already received p_x's message. If not, it waits until p_x's message is received. In either case, after successfully retrieving p_x's message, it decides for the value contained in the message and terminates its execution of the protocol by forwarding p_x's message to every correct process that have not yet received it. Algorithm 2 is the pseudo-code of the concurrent tasks that implement the asynchronous part of the protocol.

Algorithm 2 The pseudo-code of GSDP-consensus(q, p) protocol executed by process p_i

```
% shared variables
bagOfMessages_i = ∅
decided_i = false

% Proposition task
when propose(v_i) is invoked
  if i ≤ p then send m_i(v_i) to all processes end if
end when
||
% Listening task
while not decided_i do
  when receive m_j(v_j) from p_j
    if m_j(v_j) not in bagOfMessages_i then
      add m_j(v_j) to bagOfMessages_i
      notify gsdp_i of the reception of a proposition message from p_j
    end if
  end when
end while
||
% Decision task
while not decided_i do
  g = getGSD()
  x = g.consensusal_identity
  if x ≠⊥ then
    waituntil m_x(v_x) in bagOfMessages_i
    m_x(v_x) = getConsensualMessage(x, bagOfMessages_i) % retrieves p_x's message
    send m_x(v_x) to all p_k such that g.detection_vector[k] = 0 ∧ g.reception_matrix[k, x] = 0
    decided_i = true
    return(v_x) % decides for the value proposed by p_x
  end if
end while
```

4 Performance Evaluation

In this section we analyze the performance of the protocols presented in this paper and compare their performance with that of other protocols previously published. Performance prediction of most consensus protocols for asynchronous systems augmented with unreliable failure detectors use a round-based model. Two metrics are normally used: i) the time complexity of the protocol, given by the number of communication rounds that the protocol needs to execute in a particular run; and ii) the message complexity, given by the number of messages that are sent by the processes executing a particular run of the protocol. In the following we compare the performance of several consensus protocols, including the GSDP-consensus(q, p) protocols, using these metrics.

In the GSDP-consensus(q, p) protocols, processes may send messages in two parts of the protocol:

when proposing their values, and, possibly, when diffusing the consensual message, just before deciding. Thus, considering the number of rounds, the protocols GSDP-consensus(q, p) require either 1 or 2 communication rounds, in the worst case. In particular, the protocols GSDP-consensus(n, p) require just a single communication round, and are, therefore, optimal with respect to this metric. It is worth noting that, like other protocols based upon strong failure detectors, the number of rounds is independent of the number of failures that actually occur during a particular run of the protocol.

The message complexity of the protocols also vary from protocol to protocol. In failure-free runs, the protocols GSDP-consensus(q, p), send $n.p$ messages in the proposition phase, while in the decision phase up to $n - q$ messages are sent by each process that decides. Thus, in failure-free runs, the message complexity of these protocols is: $n.p + n(n - q) = n(n + p - q)$. In summary, considering message complexity, the most expensive protocol is GSDP-consensus($f + 1$, n) with a cost of $n(2n - f - 1)$, while the cheapest protocol is GSDP-consensus(n, $f + 1$) with a cost of $n(f + 1)$. Finally, the message complexity of all protocols is reduced as failures occur (this is obvious, since faulty processes do not send messages).

Dutta and Guerraoui compare several indulgent consensus protocols considering a failure-free scenario and several scenarios where failures occur [9]. The consensus protocols that they propose in [9] achieve consensus in 2 rounds in all scenarios and are the most efficient among the protocols surveyed. It is important to point out that these results consider only runs in which the failure detector makes no mistake. For those runs in which the failure detector may make mistakes, the maximum number of rounds cannot be easily computed a priori. Considering message complexity, all indulgent consensus protocols require the execution of at least one reliable broadcast. Thus, the message complexity is always greater than $(n - 1)^2$.

Brasileiro *et al.* present a consensus protocol that is able to decide in a single round in "good" runs (those in which enough processes propose the same value), however, in the other runs the protocol relies on an underlying consensus protocol, therefore requiring more rounds to decide [2]. The first part of the protocol (that tries to achieve fast decision) requires n^2 messages, which is the message complexity of the protocol in "good" runs.

Protocols based on stronger failure detectors may require as much as n rounds to reach a decision. For instance, the consensus protocol that uses a failure detector of the class S presented in [5], always requires n rounds, each round requiring n^2 messages. Thus, in failure-free runs, the protocol requires n^3 messages to be sent.

5 Conclusion

In this paper we have detailed the global state digest provider (GSDP), an abstraction to support the implementation of distributed protocols. Informally, a *global state digest* is a bounded and consistent summarized representation of the local states of all processes that run the protocol (including their operational status: correct or crashed). It is a stronger abstraction than a perfect failure detector and, therefore, provides a more powerful framework for the design of fault-tolerant distributed protocols. Regarding the consensus problem, the main contribution of the GSDP is that it provides information that allows a consensus protocol to adapt itself to the fluctuations on the contention levels experienced by the system during a particular execution of the protocol - a feature that is present in no other consensus protocol

based upon perfect failure detectors. Nevertheless, the implementation of a GSDP requires no other assumptions than those already required to implement perfect failure detection, namely the existence of an (ideally fast) synchronous subsystem that is able to guarantee a known upper bound to the end-to-end communication delays between any two correct nodes.

Whether suitable GSDPs may be designed to support the design and implementation of other distributed protocols is an open issue that we are currently investigating. Nevertheless, we believe that a GSDP is a useful abstraction to any distributed problem that can take advantage of receiving GSDs with the following *monotonicity* property: let GSD_r be the set of all GSDs that happen in any run r of a protocol P that solves a given distributed problem, and assume that all elements in every GSD_r are totally ordered, such that for any two elements g and g' of GSD_r, either $g \rightarrow g'$ or $g' \rightarrow g$; let $\mathcal{P}$ be the set of predicates that P applies on the GSDs it is delivered on its run r, and g_1 and g_2 two elements of GSD_r, such that $g_1 \rightarrow g_2$; GSD_r is monotonic in relation to P iff $\forall p \in \mathcal{P}$, if p holds in g_1 then it also holds in g_2. In a recent paper, we have shown how a cheap and safe GSDP service could be built using off-the-shelf components such as Linux boxes connected via a switched Fast-Ethernet links [4].

References

[1] AGUILERA, M. K., LANN, G. L., AND TOUEG, S. On the impact of fast failure detectors on real-time fault-tolerant systems. In 16^{th} *International Symposium on Distributed Computing (DISC 2002)* (Tolouse, France, October 2002), pp. 354–369.

[2] BRASILEIRO, F. V., GREVE, F., MOSTEFAOUI, A., AND RAYNAL, M. Consensus in one communication step is possible. Research Report 1321, INRIA, 2001.

[3] BRITO, A., AND BRASILEIRO, F. Programando um subsistema síncrono para suporte a mecanismos eficientes de tolerância a falhas. In *Workshop de Tolerância a Falhas* (2004).

[4] BRITO, A., AND BRASILEIRO, F. A cheap and safe cots wormhole for local area network. In *Proceedings of the 10th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS'05)* (2005).

[5] CHANDRA, T., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*, 2 (Mar 1996), 225–267.

[6] CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. The weakest failure detector for solving consensus. *Journal of the ACM 43*, 4 (Jul 1996), 685–722.

[7] CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. Atomic broadcast: from simple message diffusion to Byzantine agreement. In *Proceedings of the* 15^{th} *IEEE International Symposium on Fault-Tolerant Computing (FTCS'85)* (Ann Arbor, Jun 1985), pp. 200–206.

[8] DOLEV, D., DWORK, C., AND STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *Journal of the ACM 34*, 1 (Jan 1987), 77–97.

[9] DUTTA, P., AND GUERRAOUI, R. Fast indulgent consensus with zero degradation. In *Proceedings of the* $4th$ *European Dependable Computing Conference (EDCC4)* (Toulouse, France, Oct 2002).

[10] DWORK, C., LYNCH, N. A., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of the ACM 35*, 2 (Apr 1988), 288–323.

[11] FISCHER, M. J. The consensus problem in unreliable distributed systems. Research Report 273, Yale University, Jun 1983.

[12] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. D. Impossibility of distributed consensus with one faulty process. *Journal of ACM 32*, 2 (Apr 1985), 374–382.

[13] HURFIN, M., AND RAYNAL, M. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing 12*, 4 (1999), 209–223.

[14] MOSTEFAOUI, A., AND RAYNAL, M. Solving consensus using chandra toueg's unreliable failure detectors: a general quorum based approach. In *Proceedings of the* 13^{th} *International Symposium on Distributed Computing (DISC'99)* (Bratislava, Slovaquia, Sep 1999), pp. 49–63.

[15] SAMPAIO, L. M. R., BRASILEIRO, F. V., DA C. CIRNE, W., AND DE FIGUEIREDO, J. C. A. How bad are wrong suspicions? towards adaptive distributed protocols. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2003)* (San Franciso, California, USA, June 2003), pp. 551–560.

[16] SCHIPER, A. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing 10*, 3 (Apr. 1997), 149–157.

[17] VERÍSSIMO, P. Uncertainty and predictability: Can they be reconciled? *Future Directions in Distributed Computing, Springer Verlag LNCS 2584* (May 2003), 108–113.

[18] VERÍSSIMO, P., AND CASIMIRO, A. The Timely Computing Base model and architecture. *Transactions on Computers - Special Issue on Asynchronous Real-Time Systems 51*, 8 (Aug. 2002).

Appendix A. Proof of Correctness of the GSDP-consensus(q, p) Protocols

Lemma 1 Every correct process that executes the protocol presented in Algorithm 2 eventually decides some value (*termination*).

Proof If a correct process ever decides it does so while executing the decision task of Algorithm 2. In this task a decision is made only if the process is delivered a GSD g, such that $g.consensual\_identity \neq \perp$. Therefore, to prove the lemma, we must first show that eventually every correct process is delivered such a GSD. From the *validity* property, a GSD g with $g.consensual\_identity \neq \perp$ can only be delivered if there is a p_x for which there are at least $q - f_{actual}$ occurrences of i, $1 \leq i \leq n$, such that $g.detection\_vector[i] = 0$ and $g.reception\_matrix[i, x] = 1$.[2] Since channels are reliable and at least p, $p \geq f + 1$, processes broadcasts their values to all processes, the proposition message of at least 1 process will be received by all processes that have not crashed, *i.e.* $n - f_{actual}$ processes. Without loss of generality, let p_x be such a process. The *strong completeness of reception* property guarantees that eventually, all GSDs delivered by every correct GSDP module have $reception\_matrix[i, x] = 1$, for every correct p_i. On the other hand, the *strong completeness of detection* property guarantees that eventually, all GSDs delivered by every correct GSDP module have $detection\_vector[j] = 1$ for every p_j that fails. Thus, eventually all GSDs are delivered such that there are $n - f_{actual}$ occurrences of i, $1 \leq i \leq n$, for which $detection\_vector[i] = 0$ and $reception\_matrix[i, x] = 1$. Since $n - f_{actual} \geq q - f_{actual}$, then one must have $decision\_identity \neq \perp$. Since the decision task keeps constantly querying the GSDP, eventually a GSD g with $g.consensual\_identity \neq \perp$ is delivered to every correct process.

To complete the proof we must show that the proposition message sent by the process whose identification is
$g.consensual\_identity$ has indeed been received by every correct process p_i, and therefore, can be retrieved from $bagOfMessages_i$. The *strong accuracy of detection* property guarantees that, for any correct process p_i, no GSDP module will ever deliver a GSD with $g.detection\_vector[i] = 1$, thus

[2] Notice that this condition does not guarantee that $g.consensual\_identity = x$, but only that $g.consensual\_identity \neq \perp$.

the correct processes are a subset of the processes that g indicates not to have failed (*i.e.* every p_i such that $g.detection\_vector[i] = 0$). The *strong accuracy of reception* property guarantees that if $g.reception\_matrix[i, x] = 1$ then p_i has indeed received p_x's message. Since $q \geq f+1$ and $f \geq f_{actual}$, then, there is at least 1 correct process among the $q - f_{actual}$ processes that have received p_x's message. This process will complete p_x's broadcast, thus even if p_x fails while broadcasting its message, it is guaranteed that every correct process will receive p_x's message. Hence the lemma. □

Lemma 2 Every process that executes the protocol presented in Algorithm 2 decides at most once (*uniform integrity*).

Proof This is trivially met by the protocol presented in Algorithm 2. As can be seen in the pseudo-code of the protocol, there is only one decision point for any process p_i that decides. Further, after deciding, a process terminates its execution of the protocol. □

Lemma 3 If a process that executes the protocol presented in Algorithm 2 decides for the value v, then v was proposed by some process (*uniform validity*).

Proof In Algorithm 2, the decision value of a process p_i is one that has been encapsulated in a message contained in $bagOfMessages_i$. The only messages that enter $bagOfMessages_i$ of a process p_i are the proposition messages sent by the processes executing the protocol. Hence the lemma. □

Lemma 4 No two processes that execute the protocol presented in Algorithm 2 decide differently (*uniform agreement*).

Proof Lemma 1 shows that every correct process decides. Let p_x be the process whose proposition is the decision value of some correct p_i. Thus, p_i must have been delivered a GSD g such that $g.consensual\_identity = x$. From the *write-once* property if any other process is delivered a GSD g' with $g'.consensual\_identity \neq \perp$ then it must have $g'.consensual\_identity = x$. Thus, any process that decides must also decide for the value proposed by p_x. Hence the lemma. □

Theorem 1 The protocol presented in Algorithm 2 solves consensus.

Proof The proof follows directly from lemmas 1, 2, 3 and 4. □