Learning from the Past for Resolving Dilemmas of Asynchrony

Paul Ezhilchelvan and Santosh Shrivastava
School of Computing Science, Newcastle University, UK
Paul.Ezhilchelvan@ncl.ac.uk

Abstract

This paper presents two design approaches to avoid many complications introduced at both user and developer levels by the FLP impossibility. The first approach is appropriate in managed hosting environments, such as datacenters, and involves offering service guarantees with tunable success probabilities and remedial actions in the unlikely scenarios. The second is appropriate in open environments and advocates building fail-signal abstractions for hosting application-level replication.

1. Introduction

In a recent paper, Aguilera and Walfish [1] critically examine the implications, from both user and developer perspectives, of using Paxos style [14] solutions for building fault-tolerant systems for datacenters and enterprise network based applications. They then lay down a convincing rationale for seeking design alternatives to considering asynchronous model augmented with imperfect fail-detectors such as ◊S [5] or Ω oracles. Two such alternative design approaches are presented here.

Unlike [1], which advocates novel ways for building perfect fail-detectors, we take a step back and examine how the war on uncertainties intrinsic to systems is waged in other areas of system engineering. It appears that several problems are routinely addressed by taking a view that there is a statistical correlation between the past and the future behavior of a system *most of the times*. This is true even when a myriad of low- to medium-probability events affect the system behavior randomly. We believe that such a view holds in managed hosting environments, such as datacenters. Our first design approach is proposed for these environments and is a refinement on our earlier proposal [9]. It advocates modeling communication and processing delays as random variables, and proactively measuring the variables and their distribution. The resulting model is called the *probabilistically synchronous model*. Considering the identical message ordering (which is equivalent to consensus [5]) as the problem of interest here, we argue that parameterized protocols can be developed in this model and probabilistic guarantees on termination and correctness can be given to the degree practicably possible. When such guarantees are not met, remedial actions need to be initiated and such actions are outlined in the context of replicated processing.

In some environments, e.g., those exposed to intrusive attacks, it is not realistic to anticipate a statistical correlation between the past and the future behavior. For such environments, we propose the second design approach which advocates using redundancy to build *fail-signal* abstractions and then solving the ordering problem.

In motivating these approaches, we will be liberally drawing from the past work, such as synchronous atomic broadcast [7], fail-stop [15], and TCP congestion control [12]. Borrowing ideas from the past work, while proposing practical alternatives, only reinforces the viability of the approaches being proposed. For example, the principles behind taking estimates for congestion control are similar to those behind our first approach. Congestion control requires congestion detection which amounts to resolving a dilemma: is the transmitted packet really lost or merely being slow? If deemed lost, packet (re)transmission rate needs to be reduced, typically halved; else, do nothing. (This is similar to resolving FLP dilemma to be described shortly.) An incorrect detection has its cost - increased latency and reduced throughput; in an ideal set-up, it should be eliminated all together.

An implicit 'proof of viability' for our first approach follows from the observation that congestion control solutions used in practice do *not seek* mistake-free congestion detection, yet they *work* acceptably well; they expend best efforts towards minimizing the probability of making mistakes and mitigate the effects of any mistakes made. The former involves continually estimating re-transmission timeout using recent round-trip-time measurements and the latter additive

increasing of transmission rate. Thus, if uncertainties at the network level can be managed well, it is not unreasonable to suppose that a similar approach pursued in managed application environments would also yield attractive dividends in practice.

The paper is structured as follows. Section 3 presents the hypothesis of statistical correlation between the past and the future, and introduces the notion of benign and malicious adversaries which prevail when the hypothesis does and does not hold respectively. The two design approaches are described in Sections 4 and 5, and Section 6 concludes the paper. Next section briefly re-visits the arguments of [1] and presents the motivation behind our work.

2. Motivation

In the synchronous model, processing and communication delays are distributed on a known range whereas in the asynchronous model, these delays have some unknown (but finite) upper bound that cannot be estimated with certainty. Realizing the former requires careful provisioning of resources and a complete prior knowledge of user environment. So, considering the asynchronous model seems appropriate when designing dependable distributed systems on an open network such as the Internet. However, the asynchronous model introduces the FLP dilemma [10]: is the remote process of interest really crashed or merely being slow? Theoretically elegant, message-ordering solutions (e.g., Paxos) handle this dilemma by ensuring correctness of ordering (*safety*) at the cost of protocol not being able to terminate until the presumed upper bound on delays is realized (when $\Diamond S$ or Ω oracles are used) or until the random choices made by processes converge (in randomized solutions).

One of the arguments in [1] against using Paxos style solutions for building real systems is that they require, rather unrealistically, the user to wait patiently, trusting on the guarantee that the termination will occur eventually. It is also pointed out that if an 'impatient' user re-issues his command, the very safety that is so painstakingly preserved by these solutions could be undermined. The first motivation behind our work is to reverse this situation when the statistical correlation holds: the user is provided with termination guarantees while conscious efforts are made to reduce safety violations into low probability events and to mitigate the effects of any violation that might occur.

The second criticism in [1] is that implementing these solutions in an asynchronous environment is hardly practicable, and 'corners are cut' inevitably by assuming the use of synchronous devices at some level; implementations, such as Chubby [3], assume clocks with *known and/or bounded* drift rates. Yet, this is a reasonable assumption; modern systems have quite reliable clocks which can be synchronized (using external means) within some known and small bound - a core assumption in real-time and embedded systems. Arbitrary clock drift is a low-probability transient. Our second motivation is that well justified use of synchronous components, where possible, should be preferred (rather than sticking with pure asynchrony, just for the sake of it). Finally, complexity of the known solutions leads to buggy implementations. Our aim is to seek solutions that are simple to implement.

3. A Hypothesis and Adversaries

We formulate a hypothesis regarding variations in system performance observed over time and then define two types of adversaries depending on whether or not an adversary permits that the hypothesis to hold. The hypothesis is that, for the same set of input events, the performance measured in the past reasonably accurately indicates the performance to unfold in near future. Differences between past measurements and future behavior are attributed mainly to environmental *perturbs* affecting the subsystem performance, and also to measurement errors to some extent.

Designers of real-time systems realize the synchronous model by controlling all possible perturbs that could afflict system performance. This is done in two ways: either eliminating perturbs altogether or estimating *a priori* the maximum impact that perturbs might have on performance. An example of perturb elimination is the MARS operating system [13], which is at the core of time-triggered technology (http://www.tttech.com/) for building fault-tolerant real-time systems, not permitting process interrupts. Having thus controlled all perturbs, the designers are in a position to establish firm bounds on system performance. Outside the realm of such carefully-engineered synchronous systems, however, uncontrolled perturbs will always be present, or at least their absence cannot be confidently ascertained. We will term all sources of perturbs whose effect on performance is not fully controlled as the *adversary*. In other words, the adversary is not fully controllable but the effects of his actions are observable.

An adversary is said to be a *benign adversary* if the hypothesis holds despite perturbs he generates and his perturbs are called the *benign perturbs*. Thus, the impact of benign perturbs on future performance can more often be anticipated based on past observations; also, it can be accounted for in our attempts at predicting system performance. Of course, when the impact is different from the anticipated, the predicted

performance will deviate from the actual. Such deviations need to be detected and the degree of deviations measured. The latter feeds into the predictive process and the former must be handled as exceptions at the higher level; in the extreme, the user may have to handle an exception.

Typical examples of perturbs generated by a benign adversary are garbage collection, spawning of new threads, arrival of high priority inputs, excessive disk writes, so on. So, managed hosting environments, such as datacenters, can be regarded to be affected only by a benign adversary. When the hypothesis does not hold, the underlying adversary is said to be *malicious*. The classical asynchronous model, characterizing an open hosting/networking environment, assumes the presence of a malicious adversary who can arbitrarily delay completion of a task or delivery of a message. Obviously, any approach that relies on past measurements cannot be pursued, even in fail-free environments. Therefore, our search for practical alternatives to deal with a malicious adversary leads us to using redundancy for reducing the adversary into a benign one. Abstractions such as fail-stop [15] and TTCB [16] are examples of pursuing such a reductionist approach. We will elaborate a similar abstraction in Section 5.

Failures. A correct node or a process behaves according to its specification. The adversary perturbs it to fail at the moment of his choosing. A malicious adversary causes failures of arbitrary nature and a benign adversary causes failures of such nature that can be predicted with high probability. In this paper, we will assume that, with probability 1, benign failures are crashes and arbitrary failures are subject to cryptographic assumptions, i.e., failures caused by a malicious adversary are authenticated Byzantine ones.

For non-fault-tolerant embedded systems, e.g., mobile phones, the adversary, including his ability to cause failures, is (regarded to be) fully controlled over the (limited) system lifetime (through testing and reliability engineering). A crash-tolerant, synchronous (or asynchronous) system assumes a fully controlled adversary (or a malicious adversary, respectively) except for his ability to cause failures which are benign. An asynchronous system tolerant of Byzantine faults, such as [4], deals with a malicious adversary.

Finally, note that if a malicious adversary at a given level is reduced to be benign (using redundancy), then measurements taken at higher levels could be used for predicting the behavior of the sub-system below the level of reduction. For example, a TMR system with a Byzantine faulty replica responds to its clients as a system with benign adversary, and predicting its future client response times based on its past responses is meaningful, while performance prediction is meaningless for a Byzantine faulty node. Thus, building abstractions that reduce the maliciousness of the adversary at one level allows performance to be predicted for the abstract system.

We present next a design approach for each adversary by considering a system of n distributed processes, p_i, $1 \leq i \leq n$, forming a group to support state machine replication (SMR). We assume that adversary is constrained not to be able to fail more than f nodes, $f < n/2$, over a presumed life-time of the system. The support offered to the SMR layer is by identically ordering requests for replicated processing.

4. Design Approach for Benign Adversary

The Philosophy. Processes periodically measure the performance of the subsystem below in terms of parameters such as delay distribution, loss probability, jitter etc. These parameters thus abstract the lower-level performance that is relevant to the functionality provided by these processes. The order protocol is designed with loss- and fault-tolerance efforts which are also parameterized. The probability Δ that the ordering is safe when accomplished within D time is analytically estimated in terms of the protocol parameters and the ones being measured.

It is assumed that Δ is fixed (to be close to 1) by the SMR layer. Prior to ordering a request, p_i evaluates values for protocol parameters and D which are appropriate to Δ and to the lower-level subsystem performance. Unsafe ordering within D can occur with probability $(1-\Delta)$. If it does, it is detected and an exception is raised. Given that our hypothesis holds and the estimation of Δ is pessimistic to account for unanticipated deviations, exceptions should be rare.

4.1. Probabilistically Synchronous model

The model is characterized as follows.

1. All correct processes have clocks synchronized within a known bound ε.
2. If a correct p_i forms a message m at time t to be sent to all other processes,
   2.1. a correct process p_j receives m with probability $(1-q)$ that is estimated in advance;
   2.2. if m is not lost for p_j, it is received at time $t+d$, where d is a random variable whose distribution $P(d > x)$ is also estimated in advance.
3. If a correct p_i sends messages m_1 *and* m_2 at times t_1 and t_2 respectively, $t_1 > t_2$, and if both messages are received by a correct p_j at t_1+d_1 and t_2+d_2 respectively, then the jitter $((t_2+d_2) - (t_1+d_1))$ is bounded by ω with a probability close to 1 and the value of ω is estimated in advance.

Realizing the model: We assume that the first property is achieved through periodic clock synchronization using a trusted external time source. The rest are by processes sending probe messages periodically, measuring round-trip-times (RTT), and estimating the parameters using the techniques employed in TCP congestion control [12]. These techniques have matured over years of use and are particularly suited here as the retransmission timeout (RTO) estimate must be small, but not too small to cause unnecessary re-transmissions which only make the existing congestion worse. Variations in RTTs are used to estimate ω and RTO estimates to decide message losses. Any acknowledgement (to probes) taking longer will be counted as a 'loss'. Thus, we account for the effects of spikes in network traffic in the estimation of q.

We note here that though a delayed message is counted as a loss in a (pessimistic) estimation of q, it is not discarded as in Timed Asynchronous protocols [6] but are received by destinations as normal ones (and used for detecting exceptions).

4.2. Order Protocol

The protocol is an extension of our earlier work [8] developed for reliable multicasting. For space reasons, we outline the order protocol and work out an analytical estimation of D for a given Δ assuming single-packet messages (which are like the probe messages). Developing a general protocol and verifying the accuracy of estimations (using simulations) are left for a future paper.

We assume that p_i invokes the protocol at time t to order m and, for simplicity, ε = 0. Central idea is to ensure that m is transmitted at least ρ times in a crash-uninterrupted manner. A transmission is crash-uninterrupted if the transmitting process does not crash until the transmission is completed. Let R be the probability that all correct processes receive m at least once. After ρ crash-uninterrupted transmissions of m, $R=\left(1-q^{\rho}\right)^{n-1}$ if transmission losses are independent.

p_i computes ρ so that R is close to 1. It then sends m (ρ+1) times to all processes at the interval of η which is set to one half of its current RTO estimate. Control fields of m contains t and p_i's estimate of D, η, ω and ρ. Each of (ρ+1) copies of m sent is numbered as $k = 0$, 1, .., ρ. The η-separation between successive transmissions is to realize the loss independence assumption in the derivation of R. If p_i is correct, m is sent (ρ+1) times and the probability of delivery will be larger than R.

To account for possible crashing of p_i, any process p_j that receives copy k of m ensures that it either receives copy ρ or itself sends copy ρ. So, after having received copy $k < \rho$, if p_j does not receive copy k+1 within (η+ω), it starts sending to all processes copy k, copy k+1, .., copy ρ, at η intervals. Since a timeout on copy k+1 triggers a sequence of transmissions *beginning with* copy k, there are at least ρ crash-uninterrupted transmissions of m, if a correct p_j receives m for any k, $0 \le k < \rho$.

Any process (including p_i) orders m at time t+D, if it has first received (some copy of) m before t+D; otherwise, it raises an 'unsafe' exception. Note that a process that crashes in (t, t+D) does not order m even if it has received m.

To avoid redundant transmissions, a suppression mechanism is in force that works based on process ranking for m with the owner p_i ranked highest. Any p_j that is transmitting m halts the activity if a higher ranked $p_{j'}$ is observed to be doing the same. Our simulations in [8] indicate that the suppression mechanism is effective in reducing message cost from $O(n^2\rho)$ to $O(n\rho)$.

4.3. Analytical Estimation

The worst-case dissemination scenario that results in the largest D is: ϕ, $\phi \le f$, processes (starting with p_i) sequentially crash, with exactly one other process receiving m due to each crash [7]. ϕ is the expectation on the number of processes to crash *during* the order protocol execution for a given m and is assumed to have been estimated.

Let d_ϕ denote the sum of ϕ independent transmission delays. The cumulative function $P(d_\phi > x)$ can be evaluated using RTT measurements. Let $F_\phi(x)$ be the probability that the first correct process in the worst case scenario receives m within t+x. Given that the last (ϕ-1) successive transmissions in the worst-case scenario are triggered by timeout (η+ω),

$F_\phi(x)$ = P(none of the ϕ successive transmissions is lost *and* $d_\phi \le x$ - (ϕ-1) (η+ω)).

$$F_\phi(x) = (1-q)^{\phi}[1 - P(d_\phi > x - (\phi - 1)(\eta + \omega))] .$$

Let $G_{n-\phi}(y)$ be the probability that all (n-ϕ) processes (assuming that all n-ϕ are correct) receive m within y time of the first correct process transmitting m.

$$G_{n-\phi}(y) = \left[1 - \prod_{k=0}^{\rho} h(y - k\eta)\right]^{n-\phi-1} \quad \text{where} \quad h(x) = q + (1-q)P(d > x).$$

Distribution of the total time for all correct processes to receive m is the convolution: $H_{\phi,n}(x) = F_\phi * G_{n-\phi}(x)$. Note that as x increases, $H_{\phi,n}(x)$ increases. Hence D is chosen to be the

smallest: $H_{\phi,n}(D) \geq \Delta$. If p_i is correct, we have $H_{\phi,n}(D) = H_{0,n}(D) = G_n(D)$.

We note that the analytical estimation assumes that all but ϕ processes are correct. When this is not the case, estimation of D becomes pessimistic, i.e., a larger value for D than necessary is chosen; further, considering the worst-case failure scenario as the norm also makes the estimation of D pessimistic. All these provisions make the value of D used larger than minimally required and hence unsafe ordering will be unlikely even when the actual network performance deviates from that predicted. Moreover, the actual value of R is larger than what is considered here when more than ρ crash-free transmissions occur - which would be the most likely case. This means that if a correct process has not ordered m, it receives an 'unsafe' exception with a probability $\geq R$.

Handling Ordering Failure

A correct p_j can receive m not before $t+D$ with a small probability $(1-\Delta)$. When this unlikely event occurs, p_j it cannot safely order m. Moreover, j^{th} replica is put in an incorrect state, if m is a write request, and its subsequent responses will not be correct. Therefore, the following client-side requirement is necessary: clients should majority-vote server's responses. A client cannot majority-vote an incorrect response so long as a majority of servers do not develop an identically incorrect state by not ordering an identical set of inputs. We believe that the likelihood of such a failure is expected to be negligibly small.

The mitigation against unsafe ordering is to enable a correct replica to detect an unsafe ordering and apply state correction. Recall that every correct p_j receives m eventually with a probability at least as large as R; so, if it has not safely ordered m, it receives an 'unsafe' exception with a high probability. State correction will require rollback and processing the old and the missed requests in the appropriate order. Rollback in turn requires that every process periodically checkpoint its state together the sequence of requests processed thereafter. To keep the log size small, a parameter C, $C >> D$, is chosen such that $1 - H_{\phi,n}(C)$ is arbitrarily small. Thus, $t+C$ defines the time instance after which a message m sent at time t is unlikely to be received and therefore receiving an 'unsafe' exception for any m sent at or before t is hardly possible after $t+C$. So, a check-point can be deleted if it relates to ordered messages that are older by C or more.

We note that handling unsafe ordering is extremely crucial to the proposed design approach and the solution outlined here illustrates that the likelihood of unsafe ordering can be minimized to arbitrarily small values and a recovery involves only local check-pointing and processing, and no message exchange. We plan to carry out investigations in real or simulated datacenter environments to establish the cost of handling order failures which in turn will demonstrate the relative merits in pursuing this design approach.

5. Approach for Malicious Adversary

The proposed approach involves building at least f processes, say p_1, p_2, .., p_f, as *fail-signal* processes. A fail-signal p_i works correctly or signals its stopping. Fail-signal construction has been detailed in [2] and requires less redundancy than Fail-Stop [15]. Briefly, a fail-signal p_i is indeed an abstraction and is composed of two (Byzantine-prone) process replicas hosted on distinct nodes connected by a dedicated link not accessible to any other node. Replicas of p_i check each other's output values and timing. Using the link, they exchange and compare their signed outputs and produce double-signed outputs on behalf of p_i. An undue delay or a mismatch detected during a check, leads to halting of processing after outputting a pre-formed, double-signed 'fail-signal' which would indicate that p_i has halted.

The fail-signal process abstraction works if its environment ignores any of its unauthentic outputs and regards it to be correct until an authentic fail-signal is received. It requires: (i) both of its constituent process replicas not fail simultaneously and, (ii) their interconnection remain synchronous. Note that the synchrony requirement or, more precisely, the validity of the time bounds being used needs to hold only when both the constituent process replicas are correct. This aspect simplifies the realization of (ii), together with the fact that the end processes, as replicas, execute the same program and also send each other whatever is received from the environment: each replica continually modifies the bound based on the local processing load. The bound estimate used by each process will be realistic so long as the adversary affects both the process replicas nearly identically.

A significant advantage in building and working with fail-signal processes, is that the FLP impossibility ceases to exist altogether [10]; the operative status of a remote process in an asynchronous system does not have to be determined by timeouts. A fail-signal process can be assumed to be operative until a signal arrives to the contrary. This feature can be leveraged to achieve asynchronous, Byzantine fault tolerant ordering by re-using asynchronous, crash-tolerant order protocols (such as Paxos).

Consider a system with f fail-signal processes and (f+1) ordinary (Byzantine-prone) ones. The system has a total of 3f+1 ordinary processes with 2f of them (additionally) engaged in pair-wise output checking and authentication. These processes can execute any known crash-tolerant, coordinator based, protocol for input ordering, so long as an ordinary p_j becomes the coordinator only after all fail-signal p_i have signaled. (Randomized protocols cannot be used due to pair-wise checking within fail-signal processes.)

Reducing Byzantine failures to fail-signal ones and then carrying out message ordering has performance advantages over straightforward Byzantine fault tolerant ordering, such as BFT in [4]. Primarily, best case ordering latency reduces from 3 asynchronous rounds to 2 asynchronous ones plus the delay for pair-wise comparison over dedicated channels [11]. In BFT-like systems, clients select f+1 identical and authentic responses out of at most 2f+1 *signed* responses. This overhead can be reduced by having (f+1) fail-signal processes and f ordinary processes, and by having clients wait for the first response from any fail-signal process. These advantages come with the cost of extra hardware (one additional node and dedicated channels) and a restriction that two nodes of a fail-signal process cannot fail simultaneously.

6. Conclusions

Several practical systems occupy the space between well-managed synchronous systems and open, asynchronous systems. It appears that there is no *one* middle-way to build all of these 'in-between' systems. Those that can be regarded to be closer to the synchronous end can be conveniently built and adaptively maintained by considering probabilistically synchronous model. The challenges in this design approach are developing parameterized protocols, deriving analytical estimations, assessing the effect of any simplifying approximations taken and measuring the system performance accurately without imposing serious overhead. We have outlined that each challenge can be addressed for a specific problem. When systems are, or are close to, asynchronous and Byzantine, fail-signal abstractions are proposed as an intermediate design step. Multi-core machines permit a fail-signal process to be implemented on a single machine with distinct cores acting as its internal replicas, at an increased risk of both cores failing at the same time. However, multi-core fail-signal nodes are an effective deterrent against increased occurrences of soft-errors due to tera-scale hardware integration.

7. References

[1] M.K. Aguilera and M. Walfish, "No time for asynchrony", Usenix Workshop on Hot Topics in Operating Systems, May 2009.

[2] F.V. Brasileiro, P.D. Ezhilchelvan, S.K. Shrivastava, N.A. Speirs and S. Tao, "Implementing Fail-Silent Nodes for Distributed Systems", *IEEE Transactions on Computers*, 45(11): pp 1226-1238, 1996.

[3] M. Burrows, "The Chubby Lock Service for Loosely Coupled Systems", In *OSDI*, pp. 335-350, 2006.

[4] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery", *ACM Transactions on Computer Systems (TOCS)*, 20(4), November 2002.

[5] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", *JACM*, 43(2), 225-267, March 1996.

[6] F.Cristian and C. Fetzer, The Timed Asynchronous Distributed System Model, In *IEEE Transactions on Parallel and Distributed Systems*, 10 (6): 642-57, June 1999.

[7] F. Cristian, H. Aghili, R. Strong and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion To Byzantine Agreement," Proc. 15th Int'l. Symp. on Fault-Tolerant Computing (FTCS), pp. 200-206, June 1985.

[8] A. Di Ferdinando, P.D.Ezhilchelvan, and I Mitrani, Design and Evaluation of a QoS-Adaptive System for Reliable Multicasting, In Proc. 23rd SRDS, pp 31-40, October 2004.

[9] P.D. Ezhilchelvan and S.K. Shrivastava, "A Model and a Design Approach to Building QoS Adaptive systems", in *Architecting Dependable Systems II*, *Lecture Notes in Computer Science*, 3069, Springer, pp, 215 – 238, 2004.

[10] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process", *Journal of the ACM*, 32(2): 374-382, April 1985.

[11] Q. Inayat and P.D. Ezhilchelvan, "A Performance Study on the Signal-On-Fail Approach to Imposing Total Order in the Streets of Byzantium", *In Proceedings of the DSN*, pp. 578-587, 25-28 June 2006.

[12] V. Jacobson and M. Karels. "Congestion Avoidance and Control", In ACM SIGCOMM Symposium on Communications Architecture and Protocol, August 1988.

[13] H. Kopetz, "*Real-Time Systems: Design Principles for Distributed Embedded Applications*", Kluwer Academic Publishers, 1997, ISBN 0-7923-9894-7.

[14] L.Lamport, "Paxos Made Simple", Distributed Computing Column, ACM SIGACT News, 32(4), pp. 51-58, Dec 2001.

[15] F. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors", *ACM Transactions on Computer Systems*, Vol. 2(2), pp. 145-154, May 1984.

[16] P. Verissimo and A. Casimiro, "The Timely Computing Base: Model and Architecture", *IEEE Transaction on Computers*, 51(8): 916-930, August 2002.