

# A Model and a Design Approach to Building QoS Adaptive Systems

*P.D. Ezhilchelvan and S. K. Shrivastava*  
*School of Computing Science, Newcastle University, UK*

## Abstract

The paper introduces a system model called the *probabilistic asynchronous* model which characterises the context in which many practical and the Internet-based applications are built. Specifically, the model regards that basic services and system components (e.g., network services) offer Quality of Service (QoS) guarantees that are *probabilistic* in nature, admitting that the guarantees may not be met on odd occasions within the remit of the stated probabilities. The model provides a basis for designing and reasoning about systems that adaptively meet QoS obligations to end users; eventual correctness need not be compromised when components fail in their QoS obligations.

## 1. Introduction

Practical networked systems are under increasing obligations to provide certain levels of Quality of Service (QoS) to end users. Approaches to distributed system designs have thus far assumed two broad classes of computational and communication models: in the synchronous model, processing and communication delays are considered to be uniformly distributed in a known range; in the asynchronous model on the other hand, delays are finite but without any assumption on the ability to deduce delay bounds or delay distribution. So, any bound on delays, deduced however judiciously, is subject to being violated.

Basing the design of a system on the synchronous model will require careful provisioning of system resources combined with a complete prior knowledge of the user environment. This approach is only suited to a restricted set of applications. A system design based on the asynchronous model can only guarantee eventual correctness, leaving QoS considerations as an after-thought. Experience has shown that QoS provisioning, like many non-functional system properties, cannot be achieved as an add-on feature, but rather should be a core objective in the design process.

We here introduce a generic system model called the *probabilistic asynchronous* model which we claim characterises the context in which many practical and the Internet-based applications are built. Specifically, our model regards that basic services and system components (e.g., network services) do meet their performance requirements most of the time, and occasionally they may not; only when they don't, they adhere to the classical asynchronous model. Our design approach will draw from, and combine probabilistic design techniques and asynchronous ones. Its objective is to render systems that adaptively meet QoS obligations to the end users when system components meet their QoS guarantees or violate them only marginally; eventual correctness is never compromised when components fail in their QoS obligations.

There are several factors that can perturb a system's ability to maintain the QoS level desired by an end user. For example, a new user may request services with some specified QoS or an existing user may dynamically request for an enhanced level of QoS for the on-going service provisioning. In these occasions, the system has to be able to evaluate if the request can be met without jeopardising the QoS commitments already in force. Thus, a QoS enabled system should essentially be able to evaluate the feasibility of QoS provisioning and, where and if possible, to adapt itself when QoS perturbations are encountered. The system in essence needs to be *QoS adaptive* in nature. The adaptation can range from adjusting the

operational parameters (e.g., reducing the level of redundancy) to, at the extreme end, deploying additional resources such as computational capacity, bandwidth and storage.

*Timeliness, throughput, and reliability* are the common QoS attributes associated with a system and will be the focus of this chapter. Intuitively, the end-to-end QoS offered at the system level, and seen by the end user, is an aggregation (of some sort) over the QoS offered by the various subsystems that make up the system. The QoS attributes mentioned earlier are not the properties which can be sought effectively as after-thoughts; rather, as the principles of design composability [16], an efficient and modular construction of a QoS-adaptive system requires that the subsystems are built to be QoS adaptive as well.

A subsystem provides certain services to other (consumer) subsystems, by making use of the services provided to it by some other (producer) subsystems. (The end-user is the ultimate consumer.) When a consumer subsystem requests an enhanced QoS requirement, a QoS-adaptive sub-system either adapts its operations to accommodate the request or evaluates the enhanced QoS which one or more producer subsystems need to provide if the consumer request has to be satisfied. The request cannot be met without additional resources if a producer subsystem cannot support the enhanced QoS. At the bottom-end of this producer-consumer chain are the subsystems that directly manage the resources themselves: communication subsystems (CS), operating systems (OS) and storage systems (SS).

It is thus obvious that building a QoS adaptive system requires that the resourceful subsystems - CS, OS and SS - offer services with well-defined QoS guarantees. We do not propose that such guarantees need to be inviolable or deterministic, since doing so will require that the resources be adequately provided for a system load anticipated and that the load never exceed what is anticipated, i.e. QoS perturbations cannot occur. Such an approach is shown to be appropriate for hard real time systems embedded in well-characterised environments (e.g., MARS system [16]). We do not intend to restrict our focus to such systems but rather that the QoS guarantees offered by the resourceful subsystems are *probabilistic* in nature, admitting that the guarantees may not be met on odd occasions within the remit of the stated probabilities. This flexibility allows our approach to cover a wider class of applications (e.g., soft real-time) that could tolerate occasional violations of the desired QoS guarantees (including applications where, a response produced very late will not be totally useless.) For example, a response delayed by 45 minutes may still be of significance to interested bidders in an e-auction process, though the e-auction system may have guaranteed a latency bound of 15 minutes with 75% probability.

The feasibility of probabilistic evaluation of latency, throughput and failure rates of OS (real-time OS in particular) and SS is long known. However, the same cannot be asserted for the CS, particularly when the Internet is concerned. A school of opinion in the literature advocates that the pattern of the Internet message traffic, also of the LAN traffic according to some, is accurately captured only by self-similar processes [17]. As per self-similar network traffic, arrivals are concentrated (i.e., occur in batches) irrespective of the smallness of granularity of the observation time. This aspect runs counter to the notion of processes being rescaled in time so that the resulting coarsified processes lose dependence and take on the properties of an independent and identically distributed (i.i.d.) sequence of random variables.

Even if we assume that the self-similar processes model network traffic fairly accurately, these processes are not amenable to a tractable analysis necessary for deriving probabilistic distributions of message latency and throughput. The underlying cause for this difficulty is the breakdown of Markovian assumptions, and also the lack of moments (e.g., an infinite mean) of a self-similar process. Asymptotic techniques are generally employed to investigate queueing behaviour which in turn is concerned with buffer overflow and packet drop probabilities. These techniques result in asymptotic bounds on the tail probability.

In order to be able to offer any CS related QoS guarantees, it is necessary therefore to be able to exercise some control over message traffic. In practical terms this requires that the CS be supported by an ISP who can reserve bandwidth and regulate the flow, and thereby

offer probabilistic guarantees on message communication. We will assume that this is the case, which in turn means that Markovian assumptions hold.

We factor this requirement and other observations into a model called the *probabilistic asynchronous* model which will allow us to build and reason about the QoS adaptive subsystems which will form the building blocks for a QoS-adaptive system. The model is presented in the next section. We then state the QoS related properties which a subsystem need to possess and indicate how these subsystems can be ‘stacked’ in a modular fashion to render a QoS adaptive system. These issues are presented also in Section 2.

Sections 2 essentially provides a framework for building QoS adaptive systems. In Section 3, we augment this framework with a set of guidelines for developing protocols necessary to implement certain low-level subsystems which dependable applications commonly need. These subsystems are: reliable multicast [3, 15], consensus, non-blocking atomic commit [13]. The protocol development is through derivation from the rich set of known protocols developed for synchronous and asynchronous models.

As a proof of concept, we derive the reliable broadcast subsystem in Section 4 and perform probabilistic evaluation in Section 5 to make it QoS adaptive. The issue of scalability is addressed in section 6.

## 2. The Model and a Design Framework

### 2.1. Probabilistic Asynchronous (PA) Model

The system is made up of nodes that communicate using the communication subsystem (CS). A node or any process hosted within it functions correctly until and unless it crashes. A node (or a process) that does not crash is said to be correct. To present the probabilistic asynchronous model, PA model for short, we will assume a global clock which is not accessible to processes.

*Processing Delays:* Within a correct node, any task that is scheduled to be executed at time  $t$ , will be executed at  $t + \pi$  where  $\pi$  is a random variable with some known distribution.

*Storage Delays:* When a correct process initiates a storage request (for storing or retrieving of data) the request will be correctly processed at  $t + \sigma$ , where  $\sigma$  is a random variable with some known distribution.

*Transmission Delays:* If a correct process  $i$  sends a message  $m$  to another correct process  $j$  at time  $t$ , then

- $m$  is delivered to  $j$  (i.e.,  $m$  arrives at the buffer of  $j$ ) with some probability  $1-q$  ( $m$  may be lost in transmission with probability  $q$ ).
- if  $m$  is not lost, it is delivered at  $t + \delta$  where  $\delta$  is a random variable with some known distribution.

If the distributions of  $\pi$ ,  $\sigma$ , and  $\delta$  are uniform with some known mean and  $q = 0$ , then the PA model refers to the well-known synchronous model which permits upper bounds on  $\pi$ ,  $\sigma$ , and  $\delta$  to be determined with certainty; a violation of this bound is to be regarded as a performance failure if the extent of violation is observable or an omission failure otherwise. The other well-known model, *viz.* the asynchronous model, allows the upper bounds on the delays  $\pi$ ,  $\sigma$ , and  $\delta$  to be finite but unknown; it permits no assumption on the ability to deduce delay bounds. For example, any bound on delays, deduced however judiciously, is subject to being violated and such a violation will not constitute a failure.

Note that the synchronous model is subsumed in, or is a special case of, the PA model. This means that any PA protocol should run correctly (not necessarily efficiently) in a synchronous system. Conversely, if a problem is unsolvable in a synchronous system, then it cannot be solved in the PA model.

Note also that in the PA model even if process  $i$  sends  $m$  to process  $j$  a finite number of times, say  $k$  times, there is a small probability ( $q^k$ ) that  $m$  is not delivered to  $j$ . Whereas, in the synchronous and the asynchronous models, the probability of a transmission failure with  $k$ ,  $k > 1$ , attempts is assumed to be nil for some finite  $k$ . (This assumption is often referred to as the bounded degree omission failures.) The bound  $k$  is regarded to be known and unknown in the synchronous and asynchronous models respectively. Furthermore, in these two deterministic models, it is usual to abstract the redundant transmissions necessary to mask losses within the ‘send’ operation<sup>1</sup> and to denote the over-all end-to-end transmission delay as  $\delta$ . The ‘send’ operation in the PA model however refers to a single, non-redundant transmission. Thus, if  $q^k$  in the PA model is taken to be zero for some  $k$  and the delay distributions are unknown but with finite support<sup>2</sup>, then the PA model becomes the asynchronous model. Table 2 below summarises these observations on  $k$  and  $\delta$ .

MODELS	Synchronous	Asynchronous	Probabilistically Asynchronous
PARAMETERS			
Bound on successive transmission losses, $k$	Known	Finite and Unknown	Random variable on $[0, \infty]$
End-to-end delay for a ‘sent’ message, $\delta$	Has a known bound	Has a finite and unknown bound	Random variable on $[0, \infty]$ with known distribution, if message not lost
Processing and Storage delays, $\pi$ and $\sigma$	Have known bounds	Have finite and unknown bounds	Random variables on $[0, \infty]$ with known distribution

**Table 1.** Relative Comparison of the Models.

## 2.2. A Framework for Building a QoS Adaptive System

The model characterises the behaviour of base subsystems CS, OS and SS which manage respectively the capacity to communicate, process and store information. These subsystems are collectively denoted as  $S_0$  in figure 1. For a QoS adaptive system to be feasible,  $S_0$  must export a QoS management interface in addition to the traditional service interface. Using this interface, a higher-level subsystem  $S_1$  can request  $S_0$  whether a specified distribution for each of the delay variables ( $\pi$ ,  $\sigma$ ,  $\delta$ ) and a specified loss probability ( $q$ ) can be supported for a given set of requirements on processing, storage and bandwidth capacities needed to support an end user requirement. If the request cannot be supported,  $S_0$  may respond with the delay distributions and the loss probability which it can currently support.

Each subsystem  $S_i$ ,  $i > 1$ , will have two components: service component ( $service_i$ ) and QoS management component ( $qos_i$ ).

- $service_i$  implements a specified service tolerating at most  $\phi$  node crashes;
- $qos_i$  evaluates the delay and throughput distributions of  $service_i$  as a function of such distributions offered by  $service_{i-1}$ . (The throughput offered by  $S_0$  is  $(1-q)$  times the message sending rate, if a buffered message is received by the

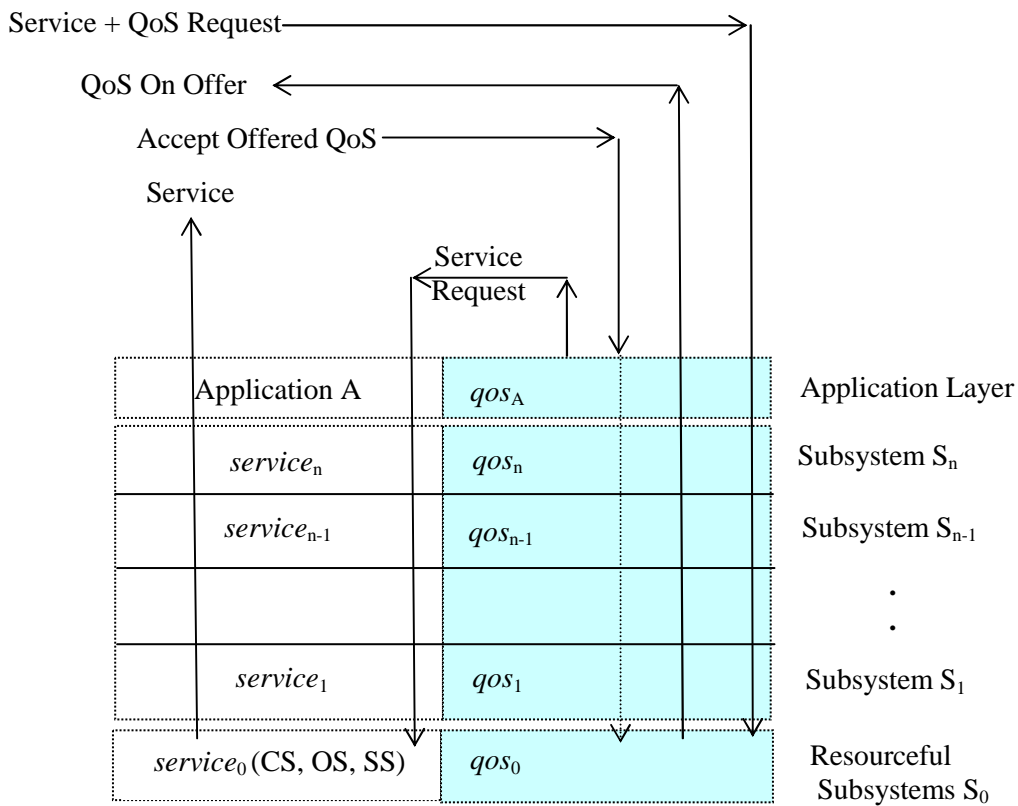
<sup>1</sup> Otherwise, correct processes cannot reliably send messages to each other, which is disallowed in both the deterministic models.

<sup>2</sup> A non-negative random variable  $\xi$  has a finite support distribution if  $P(\xi \leq x) = 1$  for some finite  $x$ .

destination process instantly.)  $qos_i$  can additionally compute the overhead that  $service_i$  would incur given the size of input from the higher level.

At the top of the stack is the application and its QoS manager. When a user submits a request with the required (probabilistic) delay and throughput guarantees, the QoS manager computes the application level overhead and the guarantees expected of  $S_n$ . The  $qos_i$  adds the overhead of  $service_i$  for this request to the application level overhead and computes the guarantees expected of  $S_{n-1}$  so that the guarantees required by the application layer can be met. The resource requirement and the expected guarantees are passed down to  $qos_{i-1}$ .

The QoS feasibility evaluation thus travels down to  $S_0$  which computes if it can maintain the necessary mean and the variance of delay distributions for the overall resource requirement. If it is possible, then the user request will be accepted; else,  $S_0$  returns the mean and variance it can sustain and the reverse computations are made by successive  $qos_i$  upwards. The user is then informed of the QoS guarantees the system can offer.



**Figure 1.** Structure of a Fault-tolerant QoS Adaptive System.

The feasibility evaluation carried out by  $qos_i$  will involve stochastic modelling and predictions of the performance by  $service_i$ . Tractable performance predictions often warrant approximations to be taken and we would propose that such approximations tend to underestimate the actual performance. This means that the system will tend to perform better than predicted, offering a better QoS than the user requested, and will thus have an inherent tendency to minimise failures on the end-to-end QoS promised.

The protocols and algorithms that are used for implementing a given  $service_i$  must be designed to preserve correctness despite node crashes and message losses, to be timely where possible, and to terminate eventually. This will enable  $service_i$  to deliver only correct responses and timely ones where possible, and also to guarantee that the response will be produced eventually if it cannot be timely enough. Design of such protocols (for  $service_i$ ) and

performance evaluation of subsystems (for  $qos_i$ ) with appropriate approximations form the core of the QoS adaptive system design.

The rest of this chapter will focus on the construction of some low-level subsystems which many fault-tolerant distributed applications are commonly known to require. (Such subsystems are collectively known in the literature as the *middleware*.) In particular, we will present guidelines for *deriving* middleware protocols suitable for this framework from the rich set of asynchronous protocols. We would like to emphasise that our guidelines do not necessarily lead to efficient PA protocols nor the derivation is *the* approach to developing PA protocols. The objective is to illustrate that the PA model forms a useful basis for QoS adaptive design and helps customisation of protocols developed in other related models.

### 3. QoS Enabled Middleware Protocol Derivation

We have earlier noted that the classical synchronous model is a special case of the PA model, with an implication that any problem that is unsolvable in the former is also not solvable in the PA model. We will therefore restrict our protocol derivation efforts to the domain of middleware services available in a synchronous, fault-tolerant system. We have also noted that the PA model with assumptions on  $q$  and finite support distributions becomes a special case of the asynchronous model. Thus, the PA middleware service protocols will be derived, where feasible, from the (deterministic) asynchronous protocols.

As a first step towards protocol derivation, we will examine the feasibility of protocol derivation. To this end, we will focus on the timing aspects of a service specification in the synchronous, asynchronous and PA models. It is fair to claim that the synchronous model specification of a given service, when expressed in the asynchronous and PA models, can differ only in the timing aspects and all other aspects must remain the same. This claim will be explained and be exemplified shortly.

Certain problems can be observed to become non-problems, when their specifications are translated from the synchronous model to the asynchronous one. This means that there is no asynchronous solution from which a PA solution can be derived; one could at best do is to attempt at *enhancing* a synchronous solution into a PA solution. Restricting ourselves to protocol derivation, we will not elaborate on protocol enhancement except to cite earlier works in the literature and briefly explain how they can be seen as enhancements of a synchronous solution in a stochastic set-up.

Among the asynchronous protocols published in the literature, we will be concerned with the ones that are based on failure detectors, FD for short [5]. FD-based protocols separate the time-related issues from time-free ones. The former are abstracted within FDs, leaving the protocols themselves time-free and deterministic. (Note that while a problem is deterministically solvable in the synchronous model, it may require certain conditions to be met for a deterministic solution to exist in the asynchronous model; failure detectors abstract these conditions.)

This separation of temporal concerns in allows an appropriate FD-based protocol to be used in the PA model relatively unchanged if efficiency considerations are temporarily set aside, provided that the required failure detector is implemented in the PA model. We identify three useful failure detectors for our purpose and propose their PA implementation. Thus, deriving a PA service protocol amounts to choosing an appropriate FD-based asynchronous service protocol and implementing the FD.

#### 3.1 Models and Service Specifications

It is not unusual to state synchronous service specifications with reference to nodes' clocks which are assumed to be synchronised within some known bound  $\epsilon$ . A node's synchronised clock is the local approximation of a global time base; in what follows, we will assume that if nodes' clocks are assumed to be synchronised in a specification, then clocks

are perfectly synchronised, i.e.,  $\varepsilon = 0$ . In other words, the synchronous specifications we consider have time expressed with reference to a global (and reliable) time base.

The timing aspect of a synchronous service specification usually runs as follows: if a specified event happens at one process at time  $t$ , then another (specified) event must occur at a correct process at  $t'$ ,  $t' \leq t + \Delta$ , where  $\Delta$  is a known constant. In the asynchronous specification of the same service,  $\Delta$  is finite but unknown: the second (specified) event must occur at a correct process *eventually*. In the PA model,  $\Delta$  is bounded by  $D$  with probability  $r_D$  that can be estimated in advance and  $r_D \rightarrow 1$  as  $D$  becomes large. Below we present the specifications of four well-known middleware services in the three models; the timing aspects shown within brackets in normal, bold, and underlined fonts refer to the synchronous, asynchronous, and PA models, respectively.

### Clock Synchronisation Service

A clock is said to be correct if its host node is correct and its running rate does not differ from that of the global reference clock by an amount whose magnitude is bounded by a small known constant  $\kappa$ : At any global time  $t$ ,

- **Precision:** The absolute difference between the readings of any two correct clocks will be within  $\varepsilon$ , (where  $\varepsilon$  is a small known constant) (**where  $\varepsilon$  is finite but unknown**) (where  $\varepsilon$  is bounded by  $E$  with probability  $r_E$  that can be estimated in advance), and
- **Accuracy:** The absolute difference between the reading of a correct clock and that of the global reference clock will be within  $\beta$ , (where  $\beta$  is a small, known constant) (**where  $\beta$  is finite but unknown**) (where  $\beta$  is bounded by  $B$  with probability  $r_B$  that can be estimated in advance).

### Atomic Commit Service

Among a set of  $n > 2$  processes that are known to each other, one process is designated as the General [18] and others as Lieutenants (which are only crash prone). The General is to broadcast a value at time  $t$  (that is known to the Lieutenants) (**that is unknown to the Lieutenants**) (that is known to the Lieutenants with a certainty that can be estimated in advance).

- **Termination:** every correct process decides at  $t + \Delta$  (where  $\Delta$  is a known constant) (**where  $\Delta$  is finite but unknown**) (where  $\Delta$  is bounded by  $D$  with probability  $r_D$  that can be estimated in advance)
- **Validity:** If the General is correct and broadcasts value  $v$ , then all correct processes decide on  $v$
- **Unanimity:** all correct processes decide on the same value (even if the General crashes at or before  $t$ )
- **Uniform Integrity:** every process decides at most once (i.e., a process that decides, decides irreversibly).

### Consensus Service

In a set of  $n > 2$  processes that are known to each other, each process is to propose a value by time  $t$  (that is known) (**that is unknown**) (that is known with certainty that can be estimated in advance).

- **Termination:** every correct process decides on some value by  $t + \Delta$  (where  $\Delta$  is a known constant) (**where  $\Delta$  is finite but unknown**) (where  $\Delta$  is bounded by  $D$  with probability  $r_D$  that can be estimated in advance),
- **Validity:** If a process decide on value  $v$ , then some process proposed  $v$ ,

- **Unanimity**: no two correct processes decide differently, and
- **Uniform Integrity**: every process decides at most once.

### Reliable Broadcast Service

In a set of  $n > 2$  processes that are known to each other, any process can broadcast a message at any time.

- **Termination**: if a correct process broadcasts a message  $m$  at time  $t$ , then all correct processes deliver  $m$  by time  $t + \Delta$  (where  $\Delta$  is a known constant) (**where  $\Delta$  is finite but unknown**) (where  $\Delta$  is bounded by  $D$  with probability  $r_D$  that can be estimated in advance)
- **Unanimity**: if a correct process delivers a message  $m$  at time  $t$  then all other correct processes deliver  $m$  by time  $t + \Sigma$  (where  $\Sigma$  is a known constant) (**where  $\Sigma$  is finite but unknown**) (where  $\Sigma$  is bounded by  $S$  with probability  $u_S$  that can be estimated in advance), and
- **Uniform Integrity**: every process delivers any given message  $m$  at most once.

A closer look at the asynchronous specification of the clock synchronisation service indicates that the underlying problem is indeed a non-problem: any two correct clocks that were initially synchronised within some finite (but unknown)  $\varepsilon_0$  and  $\beta_0$  at time  $t_0$  will remain synchronised within some finite  $\varepsilon$  and  $\beta$  over a period of any finite duration as the relative drift rates are small and bounded.

We refer the reader to Cristian's probabilistic protocol [8] which meets the PA specification of the clock service. Cristian also describes how the core aspects of his design relate to, and are enhancements over, those of the well-known synchronous protocols. For the sake of completeness, we will highlight these aspects very briefly. When a correct process reads a correct remote clock, the resulting read error can be deterministically bounded in the synchronous model. In the PA model, however, message delays are not bounded with absolute certainty. Therefore a process is required to read a remote clock more than once in a given re-synchronisation attempt; it discards those readings which are deemed to be excessively delayed and consider only those attempts in which read error is likely to be within a desired bound. A process gives up reading a clock after a finite number of attempts. If  $p$  is the probability that an attempt to read a correct clock fails, then, by Bernoulli's law, the average number of read attempts needed for successful reading of a clock will be  $(1-p)^{-1}$ . Both the synchronous and probabilistic solutions involve fault-tolerant attempts aimed at minimising the effect of an incorrect clock being read inconsistently by correct clocks. Improvements over [8] are claimed in [1].

## 3.2 Failure Detectors and Suspectors

We have observed that a class of synchronous problems can reduce to non-problems when their specifications are adopted to the asynchronous model. For the remaining set of problems, one could obtain at most three classes of detector-based asynchronous protocols which can solve their asynchronous versions. We will denote these protocol classes as C1, C2 and C3 and distinguish them by the kind of failure detectors [5] they employ:

- *perfect failure detectors* (C1 protocols),
- *eventually strong failure detectors* (C2 protocols), and
- *failure suspects* (C3 protocols)

The failure suspector of a node monitors the operational status of remote processes and regularly reports its suspicions to the local process. Its suspicions need not be accurate and may change with time; for example, after having suspected a remote process to be

crashed, a suspector may later suspect that process to be operative. However, it is not a trivial component and the suspector of a correct node has the following *completeness* property:

- If a remote process is crashed, there is a time after which the suspector permanently suspects the crashed process.

Failure detectors encapsulate failure suspectors and, additionally, detectors of a given type are an abstract representation of certain requirements imposed on, or desired of, the communication subsystem behaviour. Two types of detectors are of interest here.

A perfect failure detector of a correct node inherits the *completeness* property (of the failure suspector it encapsulates) and also satisfies the following *strong accuracy* property:

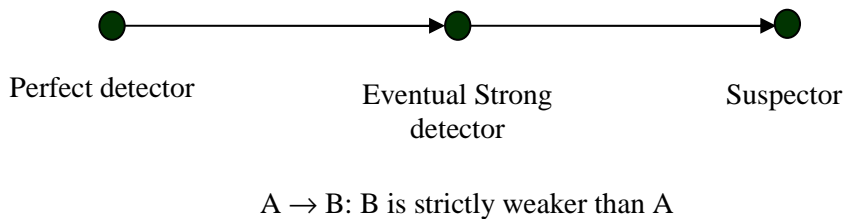
- No correct process is ever suspected.

An eventually strong failure detector has the *completeness* property and the following *eventual weak accuracy* which is expressed in a context of distributed processes monitoring each other's operative status:

- There is a time after which some correct process is never suspected by the suspector of any correct process

Note that the time after which inaccurate suspicions cease cannot be known and that the unsuspected correct process need not know that it is no longer being suspected by any one.

Chandra and Toueg, who first defined the above two and two other classes of failure detectors, also prove that a failure detector of any other class is strictly weaker than a perfect failure detector [5] (since the latter can be reduced to the former). Furthermore, it is obvious that a failure suspector that can make any number of mistakes is strictly weaker than an eventual strong failure detector. This 'strictly weaker than' relation is transitive and figure 2 indicates how the detectors and suspectors of our interest are related.



**Figure 2.** Relationship Between the Detectors.

The relation depicted in figure 2 implies that if a problem is solvable with a failure suspector then it can be solved with a perfect or eventual strong failure detector and such solutions are implementable if the failure detector can be built; similarly, if solvable with an eventual strong failure detector, then a solution with perfect failure detector is also possible.

Certain services, such as the atomic commit, require the perfect failure detector as the minimum requirement. Such services can only be solved by C1 type protocols. For certain other services, such as the consensus, the eventual strong failure detector is known to be the minimum requirement [6]. These services can have C1 or C2 type protocols. Services, such as reliable broadcast, uniform reliable broadcast, and causal broadcast, can be solved just with failure suspectors, and their solutions can therefore be of any type.

### 3.3 PA Implementations of Suspectors and Detectors

The FD-based protocols of types C1, C2, and C3 are deterministic and time-free: a protocol process responds to receiving a message or to receiving an output from the FD. Two mechanisms are needed to make these protocols work in the PA model satisfying the PA versions of their respective service specifications: the probability of service not being provided decreases with time.

- A PA version of the FD needs to be used, and
- A mechanism is needed to maximise the probability that a correct process receives the message sent by another correct process.

The second mechanism caters for the asynchronous model characteristic that the correct processes can send messages to each other successfully (i.e., with the delivery probability being 1). This mechanism and the FDs can be implemented using heart-beat messages.

**Heart-beats:** Every message sent by the protocol process is stored in the disk. A *gossiper* process periodically broadcasts the history of sent messages stored in the disk; it also requests for a retransmission if a heart-beat message received indicates that a sent message is found not received locally; and, it re-transmits the messages requested by remote processes. The longer a sent message is gossiped (before being garbage collected from the disk), the larger is the probability that all correct processes receive that message.

**Failure Suspectors:** They are the easiest ones to implement as they are allowed to make mistakes in an arbitrary manner. However, we propose that they are implemented judiciously by taking advantage of the known delay distributions of the PA model so that the mistake rate is small. For example, when message transmission delays are exponentially distributed with mean  $d$ , the transmission delay between a source and a destination is less than  $\eta$  with a probability  $\eta$  (that can be made reasonably close to 1) if  $\eta = -d \ln(1-\alpha)$ .

A simple timeout-based implementation of suspectors can be achieved as follows. Suppose that the *gossiper* process of each node transmits a ‘heart-beat’ message once every  $\tau$  time (measured as per the respective local clock). If a heart-beat message is not received from an unsuspected process  $j$  within  $(\tau+\eta)$  time, then the suspector of process  $i$  suspects process  $j$ ; if a heart-beat message is received from a suspected process, the suspicion is reversed.

**Eventual Strong Failure Detector:** Using an algorithm similar to the failure suspector, Chen *et.al.*, implement a failure detector with configurable QoS properties [7]. With known delay distributions of the PA model, one can set appropriate parameters to obtain the accuracy necessary and implement a detector that meets the specifications of eventual strong failure detector with the desired probability.

**Perfect Failure Detectors:** They cannot be implemented within an asynchronous or a PA system; an external observer who can accurately determine the operative status of processes is needed. However, by adding the following delay assumption into the PA model, we can obtain an implementation:

- *Crash-recovery delay:* a crashed process resumes functioning after a delay which is a random variable with known distribution.

The above assumption requires that a crashed node is re-booted and the pre-crash states are available in a crash-proof store. Since a crashed node recovers and its gossiper process resumes broadcasting heart-beats, a crash need not be suspected by any other process. Thus, the perfect failure detector effectively has to neither suspect nor reverse its suspicions; i.e., it is an empty module that makes no output at all.

## 4. Customising a Reliable Broadcast Service

In this section, we consider a protocol [11] that implements reliable broadcast service in a synchronous system and customise its asynchronous version for implementing the corresponding service in a PA system. The protocol customisation will also incorporate efficiency measures for reducing message overhead and these measures will be clearly indicated. For brevity, we will present the synchronous protocol in a slightly altered form to the original, but this does not alter the fundamentals.

### 4.1 The Synchronous Protocol

The message communication, invoked by the  $send(m)$  operation, can experience a uniformly distributed delay on  $(0, 2d)$ , i.e., with mean  $d$ . We assume that a perfect failure detector is available. Such a failure detector can be easily constructed since the delays are bounded by  $2d$ : if a heart-beat does not arrive for every  $(\tau+4d)$  time, the source process must have crashed.

A process that wishes to initiate a reliable broadcast of message  $m$  executes the primitive  $RBcast(m)$  and is denoted as the *originator* of  $m$ . The protocol has two features to attain reliability against process crashes resulting in a partial broadcast.

- a) The execution of  $RBcast(m)$  comprises two invocations of a  $broadcast(m)$  operation. Each of these invocations concurrently sends  $m$  once to each destination process.
- b) The responsibility for invoking  $broadcast(m)$  initially rests with the *originator* of the message, but may devolve to other processes in consequence of the *originator* crashes.

These features can be described as *Redundancy* and *Responsiveness*, respectively.

*Redundancy* of the protocol is expressed by two parameters:

- the integer  $\rho = 1$  specifies the level of redundancy; the originator of  $m$  makes  $\rho + 1$  attempts to broadcast  $m$  (if operative); these attempts are numbered 0 and 1.
- the interval between consecutive broadcasts is of fixed length,  $\eta = d$ .

*Responsiveness*: If the originator of a message crashes during its redundant broadcast attempt, the destination processes respond by taking over the broadcasting responsibility upon themselves. To facilitate this takeover, each copy of a message,  $m$ , has fields  $m.copy$ ,  $m.originator$  and  $m.broadcaster$ ; these specify the number of the current broadcast attempt (0 or 1), the index of the originating process, and the index of the process that actually broadcast the message  $m$ , respectively. The values of  $m.originator$  and  $m.broadcaster$  will be different if a destination process carries out the broadcasting of  $m$ .

When a destination process receives a message  $m$  for the first time, it delivers  $m$ ; if  $m.copy = 0$ , it must be prepared to become a broadcaster of  $m$  if necessary. It waits to receive  $m.copy = 1$  to be received; while waiting if its failure detector suspects  $m.broadcaster$ , then it executes  $RBcast(m)$ .

The protocol tolerates the crash of the originator and any number of destination processes: a correct destination process that delivers  $m$  either carries out  $RBcast(m)$  or receives  $m.copy = 1$ ; in the latter case,  $m.broadcaster$  did not crash while broadcasting  $m.copy = 0$  and every other correct process must deliver  $m$ . Further, it requires only two broadcasts if the originator is correct which is the optimal requirement for a deterministic reliable broadcast protocol.

## 4.2 A Probabilistic Protocol

Deriving a probabilistic protocol will require implementing the PA version of the perfect failure detector. The following observations can be made on the resulting protocol:

The number of broadcasts carried out will always be two – both by the originator, since no destination process ever suspects an originator crash; this means that a correct destination may not receive  $m$  directly from the originator process with probability  $q^2$ .

The gossip process plays a role in increasing the probability that  $m$  is delivered to all correct destination processes.

The latency distribution will be influenced by the probability that (i) the originator crashes while broadcasting  $m$ ,  $copy = 0$  and the recovery delay distribution; and (ii) a crashed destination process recovers before the gossip of the originator node discards  $m$ .

Implementing perfect failure detector in a PA system requires an additional assumption that a crashed process recovers. This assumption and the dependency of latency distribution on the recovery delays may be inappropriate in some practical contexts, in particular, when these requirements can be avoided. We recall that the reliable broadcast problem is also solvable just with a failure suspector. So, in what follows, we derive a probabilistic protocol using a failure suspector which will not require a crashed node to recover (i.e., if a crashed process recovers, it is assumed to have a new identifier). In this derivation, we will also incorporate adaptive measures by letting  $\rho$  be chosen appropriately rather than restricting it to two, and efficiency measures by (i) suspecting failures using the broadcast messages in addition to on heart-beats, and (ii) adding a third feature (called the *Selection*). It is assumed that the numbering of processes implies a ‘seniority’ ordering: process  $i$  is said to be ‘more senior’ than process  $j$  if  $i < j$ .

## 4.3. A Practical Probabilistic Protocol

The reliable multicast protocol has three features which are designed to assure high probability of success at tolerable cost in message traffic:

- a) The execution of  $RBcast(m)$  comprises more than one invocation of a  $broadcast(m)$  operation. Each of these invocations concurrently sends the message  $m$  once to each destination.
- b) The responsibility for invoking  $broadcast(m)$  initially rests with the originator of the message, but may devolve to another process, and then to another, in consequence of crashes, message losses or excessive delays.
- c) In the event of such a devolution, a decision procedure attempts to select exactly one process to take over the broadcasting responsibilities.

The last feature is termed as the *Selection* while the first two are as *Redundancy*, *Responsiveness* as before.

*Redundancy*. The redundancy of the protocol is controlled by two parameters:

- The integer,  $\rho$ , now specifies the level of redundancy desired; the originator of a message makes  $\rho + 1$  attempts to broadcast it (if operative); these attempts are numbered  $0, 1, \dots, \rho$ ; typically,  $\rho \geq 1$ .
- The interval between consecutive broadcasts is of fixed length,  $\eta$ ; that length is chosen to be as small as possible, but sufficiently large to make any dependencies between consecutive broadcasts negligible.

One way of choosing  $\eta$  is to require that the transmission delay between a source and a destination is less than  $\eta$  with a given probability,  $\alpha$  (reasonably close to 1). In the case of exponentially distributed delays with mean  $d$ ,  $\eta$  is given by  $\eta = -d \ln(1 - \alpha)$ .

*Responsiveness.* Every process that receives a message,  $m$ , such that  $m.copy = k < \rho$ , must be prepared to become a broadcaster of  $m$  if necessary. It does so by setting a timeout interval of length  $\eta + \omega$ , with some suitable value of  $\omega$  ( $\eta$  is the interval between consecutive broadcasts, while  $\omega$  accounts for differences in transmission delays, or ‘jitter’). If copy  $k+1$  of  $m$  arrives from the broadcaster of copy  $k$  before the timeout expires, then all is well with that broadcaster; the receiver process sets a new timeout of  $\eta + \omega$  for the next copy (if there is one). Otherwise, the receiver suspects that the process  $m.broadcaster$  has crashed while broadcasting copy  $k$  of  $m$ , and that it is the only process to have received any copy of  $m$ . It therefore prepares to appoint itself as a broadcaster of copies  $k, k+1, \dots, \rho$ .

However,  $m.broadcaster$  may not in fact have crashed; copy  $k+1$  of  $m$  may just be delayed unduly or lost; moreover, even if  $m.broadcaster$  has crashed, this receiver may not be the only process that has observed the crash. In order to avoid multiple receivers becoming broadcasters unnecessarily, a further random wait,  $\zeta$ , uniformly distributed on  $(0, \eta)$ , is added to the timeout interval  $\eta + \omega$ . If a copy number  $k$  or higher is not received before the expiration of  $\zeta$ , this receiver appoints itself as a broadcaster. Otherwise it sets a new timeout of  $\eta + \omega$ .

*Selection.* The protocol guards against multiple self-appointed broadcasters (an efficiency measure). It requires that any broadcaster with index  $i$ , whose latest broadcast has been of copy  $k$  of the message, should relinquish its broadcasting role in any of the following circumstances:

Process  $i$  receives a message  $m$  such that  $m.copy = k$  and either  $m.broadcaster < i$  or  $m.broadcaster = m.originator$ . That is, a more senior process has assumed the duties of broadcaster, or the originator has not in fact crashed.

Process  $i$  receives a message  $m$  such that  $m.copy > k$ . This would happen if process  $i$  has missed one or more copies of  $m$ , and now learns that another broadcaster is closer to completing the protocol.

Suppose that a process which abandoned its broadcasting role and has set a timeout expecting a copy, say,  $k$ , from a given broadcaster. It will have to reset that timeout if either copy  $k$  is received later from a broadcaster more senior to the current broadcaster or from the originator, or copy  $k+1$  or higher is received from any broadcaster. This is necessary because when the first broadcaster receives the message which this process just received, it would relinquish its broadcasting role.

The purpose of these provisions is to avoid unnecessary broadcasts and hence message traffic, while ensuring that  $\rho + 1$  copies of each message are broadcast. The idea is that when any broadcaster crashes, all receivers that timeout on  $\eta + \omega + \zeta$  will briefly become broadcasters, but after that only one is most likely to continue broadcasting, at intervals of length  $\eta$ . That process will be a receiver process if the originator has crashed or its messages suffer excessive delays. A detailed pseudo-code description of the reliable multicast protocol executed by the process  $i$  is shown in Figure 3. (The variables  $max\_recd_i$  and  $leader_i$  denote the number of the largest copy of the message received, and the index of the broadcaster from which the next copy is expected, respectively).

```

RBCast(m)
1)  m.originator ← i ; m.SequenceNo ← seq_number ;
2)  m.copy ← 0 ;
3)  repeat ρ + 1 times :
4)  { broadcast(m) ; wait(η) ; m.copy ← m.copy + 1 ; }

```

```

RMDeliver()
begin
  cobegin // message-handling part
5)   receive(m);
6)   if new(m) then
7)     begin
8)       max_recdi(m) ← m.copy;
9)       leaderi(m) ← m.broadcaster;
10)      last_own_bcasti(m) ← -1;
11)      deliver(m); // m is delivered (once)
12)    end

13)   if (m.copy = ρ) then { max_recdi(m) ← MaxInt; }

14)   if(m.copy > max_recdi(m)) or
15)     (m.copy=max_recdi(m)and(m.broadcaster=m.originator
or m.broadcaster < leaderi(m)) then
16)     begin
17)       max_recdi(m) ← m.copy;
18)       leaderi(m) ← m.broadcaster;
19)       set timeout for η+ω;
20)     end
  coend
  cobegin
    // timeout-triggered, timer-driven part
    timeout(m) ⇨
    begin
21)      leaderi(m) ← MaxInt;
22)      wait(ζ) ;
23)      if leaderi(m) = MaxInt then
24)        {leaderi(m) ← i; create_thread Broadcaster(m)};
    end
  coend
end
-----
Broadcaster(m)
begin
25) while((max_recdi(m) < ρ) and (leaderi(m) = i))
    do
26)   m.copy ← max{last_own_bcasti(m) + 1, max_recdi(m)};
27)   broadcast(m);
28)   max_recdi(m) ← m.copy;
29)   last_own_bcasti(m) ← m.copy;
30)   wait(ζ)
31) od
32) die; // the thread dies.
end

```

**Figure 3.** Pseudo-code description of the protocol.

Referring to Figure 3, an execution of  $RBCast(m)$  involves the originator setting the fields  $m.originator$  and  $m.sequenceNo$  and, as described earlier, making  $(\rho + 1)$  invocations of  $broadcast(m)$  with  $m.copy$  increasing from 0, 1, .. ,  $\rho + 1$ . The primitive  $broadcast(m)$  sets the  $m.broadcaster$  field and concurrently sends  $m$  to all other processes in the group. The protocol for delivering a reliable multicast message is  $RMDeliver()$ , and is structured into two concurrently executed parts.

The first part handles a received message and the second part the expiry of timeout ( $\eta + \omega$ ). Three integer variables are maintained for a received message  $m$  distinguished by  $\{m.originator, m.sequenceNo\}$ :

- $max\_recd_i(m)$  has the largest copy number received for  $m$ .
- $leader_i(m)$  has the index of the process from which  $m$  with copy  $max\_recd_i(m) + 1$  is expected.
- $last\_own\_bcast_i(m)$  contains the copy number of  $m$  which the process  $i$  broadcast when it last acted as a self-appointed broadcaster.

A received message calls for one or more of the following three actions:

- New  $m$ . Variables are initialised and  $m$  is delivered (lines 6-12).
- $m.copy = \rho$ . Blocks any future occurrence of the third action (described next), by setting  $max\_recd_i(m)$  to  $\infty$  ( $MaxInt$ ) (line 13). Note that a new  $m$  can have  $m.copy = \rho$  if earlier copies are lost or excessively delayed.
- Change of  $leader_i(m)$ . The received  $m$  indicates one of the circumstances (described earlier) in which the process  $i$  needs to either relinquish its broadcasting role or change the broadcaster from which the next copy is expected. A new timeout ( $\eta + \omega$ ) is set after  $max\_recd_i(m)$  and  $leader_i(m)$  are updated (lines 14-20).

When timeout ( $\eta + \omega$ ) for  $m$  expires, an additional timeout  $\zeta$  is set, during which a message with appropriate copy number from any broadcaster is admissible. So,  $leader_i(m)$  is set to  $MaxInt$  (line 21). If no such message is received, process  $i$  appoints itself as a broadcaster and sets up a thread  $Broadcaster(m)$  (lines 22-24).

The thread  $Broadcaster(m)$  broadcasts  $m$  only if the process  $i$  remains to be the broadcaster (i.e.,  $leader_i(m) = i$ ) as per selection rule; otherwise, it dies (lines 25-32).

## 5. QoS Evaluation of the Protocol

The QoS evaluation will involve establishing the *eventual delivery probability* and the probability distributions of two delays: *latency* and *relative latency* which are defined as follows.

*Eventual Delivery*: An invocation of  $RBCast(m)$  delivers  $m$  to all correct destination processes with a probability which can be made arbitrarily close to 1.

*Latency Bound*: The interval between an invocation of  $RBCast(m)$  and the first instant thereafter when all correct destination processes have received  $m$ , does not exceed a given latency bound,  $D$ , with a probability,  $r_D$ , which can be evaluated in advance.

*Relative Latency Bound or Skew*: The instants when  $m$  arrives for the first time at every operative destination process are within an interval of a given length,  $S$ , with a probability,  $u_S$ , which can also be evaluated in advance.

Note that the eventual delivery probability is  $r_D$  as  $D \rightarrow \infty$ . In what follows, we will be concerned with deriving only the delay distributions. We will make the following approximation. In estimating  $r_D$ , the originating process is assumed not to crash. This is not unreasonable because it will generally be a pessimistic approximation: if the originator crashes at some point after broadcasting copy 0 but before broadcasting copy  $\rho$ , some of the processes that receive a copy of  $m$  will make at least one broadcast themselves. Thus, the number of broadcasters and hence the probability of eventual delivery will increase. Of course it is possible that the originator crashes during broadcast 0, and no operative process receives any copy of  $m$ ; we consider the probability of that event to be negligible.

## 5.1 Latency

Let  $\xi$  be the random variable representing the execution time of a  $send(m)$  operation, i.e., the transmission time of a message from a given source to a given destination. The probability,  $h(x)$ , that such an operation *does not* succeed within time  $x$ , is equal to

$$h(x) = q + (1 - q)P(\xi > x), \quad (1)$$

where  $q$  is the probability that the message is lost. By definition,  $h(x) = 1$  if  $x \leq 0$ . In the case of exponentially distributed transmission times (with mean  $d$ ), the above expression becomes

$$h(x) = q + (1 - q)e^{-x/d}, \quad (2)$$

and  $h(x)=1$  for  $x \leq 0$ .

Since the originator makes its  $k^{\text{th}}$  broadcast at time  $k\eta$  ( $k = 0, 1, \dots, \rho$ ), the probability,  $g_D$ , that a *given destination* does not receive any of the  $\rho + 1$  copies within time  $D$ , is given by

$$g_D = \prod_{k=0}^{\rho} h(D - k\eta). \quad (3)$$

Hence, the probability,  $r_D$ , that every destination receives at least one copy of the message within an interval of length  $D$  is equal to

$$r_D = (1 - g_D)^{n-1}. \quad (4)$$

If some of the destinations have crashed, then (4) is an underestimate of the probability that all operative destinations receive at least one copy within time  $D$ . This is so because the term  $(1 - g_D)$  would then be raised to a lower power, which would make the resulting probability larger.

A user requirement, stated in terms of a success probability  $R$  and latency  $D$ , is achievable if the probability evaluated by (4) satisfies  $r_D \geq R$ ; otherwise it is not achievable.

## 5.2 Relative latency

Suppose now that at a given moment,  $t$ , a given process,  $p_i$  (different from the originator), receives copy number  $k$  of the message. Of interest is the probability,  $u_k(S)$ , that all other processes will receive at least one copy of the message with relative latency  $S$ , i.e., before time  $t+S$ .

The implication of  $p_i$  receiving copy number  $k$  is that the originator has started broadcasting no later than at time  $t - k\eta$  in the past, and has issued at least  $k$  broadcasts. Consider a given process,  $p_j$ , different from the originator and from  $p_i$  as well. The probability,  $g_k(S)$ , that  $p_j$  will not receive any of those  $k$  copies before time  $t+S$  is no greater than

$$g_k(S) = \prod_{m=0}^k h(S + m\eta), \quad (5)$$

where  $h(x)$  is given by (1). In addition, if  $k < \rho$ ,  $p_j$  may receive copies  $k, k+1, \dots, \rho$ , from  $p_i$ , in the event of the originator crashing. Those latter broadcasts would be issued at times  $t + \eta + \omega + \zeta, t + 2\eta + \omega + \zeta, \dots, t + (\rho - k + 1)\eta + \omega + \zeta$ , assuming that no other process starts broadcasting. Since  $\zeta$  is uniformly distributed on  $(0, \eta)$ , we can pessimistically replace  $\zeta$  by  $\eta$ . The probability,  $\tilde{g}_k(S)$ , that  $p_j$  will not receive any of the messages from  $p_i$  before time  $t+S$  is thus approximated by

$$\tilde{g}_k(S) = \prod_{m=1}^{\rho-k+1} h(S - (m+1)\eta - \omega), \quad (6)$$

where  $\tilde{g}_\rho(S) = 1$  by definition; also,  $h(x) = 1$  if  $x \leq 0$ .

Thus, a pessimistic estimate for the conditional probability,  $u_k(S)$ , that all other processes will receive at least one copy of the message with relative latency  $S$ , given that a given process has received copy number  $k$ , is given by

$$u_k(S) = [1 - g_k(S)\tilde{g}_k(S)]^{n-2}. \quad (7)$$

A pessimistic estimate for the conditional probability,  $u_k(S)$ , that all other processes will receive at least one copy of the message with relative latency  $S$ , given that a given process has received any copy, is obtained by taking the smallest of the above probabilities:

$$u_S = \min(u_0(S), u_1(S), \dots, u_\rho(S)). \quad (8)$$

This quantity may be used in deciding whether a user requirement, stated in terms of a desired success probability  $U$  and relative latency  $S$ , is achievable or not: the requirement is achievable if  $u_S \geq U$ .

## 6. Scalability

An observation which is gaining widespread acceptance is that the FD protocols and also their close cousins – virtually synchronous protocols – cannot scale while maintaining throughput simultaneously. It was borne out of a variety of experiments by Gupta et. al. who also provide informed explanations for it [14]. These explanations are of interest here as we propose to derive PA protocols from the FD-based ones.

The central argument for the non-attainability of a scalable throughput is that the probability of certain assumptions being incorrect increases, as the system size grows, from being negligible to being significant, causing the background overhead to rise and the throughput to become unstable. For example, it is common to model a correct process operating continuously in the steady state, yet events such as page faults, according low scheduling priority, etc., can cause a process to suspend its operation (i.e., to go to ‘sleep’) for a while. As the number of processes increases, the probability of at least one process being in sleep mode at any given instance of time, rises to a significant level. Since the asynchronous protocols offer deterministic guarantees, correct processes need to retain each message until all processes have acknowledged it. With at least one process being slow at any given time, the buffers can easily overflow causing high loss rates or the flow control to kick in.

PA protocols, however, offer only probabilistic guarantees and can well avoid suffering from this phenomenon to a considerable extent, and go on to offer a scalable throughput. The RBCast( $m$ ), for example, broadcasts  $m$  only for  $\rho$  times and the gossip process afterwards retains the sent messages only for a bounded number of gossip rounds (see subsection 3.3). Thus, the protocol should not suffer from one or more processes being asleep at any time. We suspect, on the other hand, that the  $\zeta$ -timer introduced as an efficient measure (see *Selection* in subsection 4.3) may suffer from the effect of a low probability event becoming a significant one due to increase in system size. This  $\zeta$ -timer mechanism traces its origin to the *random assessment delay* (RAD) of the SRM protocol [12] which is a best-effort reliable multicast protocol. Gupta et. al., observe that the RAD is ineffective as the size increases and offers smaller throughput than what can be achieved without it, and advocate *randomised gossip* [2] as a solution to achieve scalable throughput. One could achieve an efficient scalable measure by adopting this technique, for example, by designing a self-appointed broadcaster to broadcast message  $m$  only to a subset of randomly selected receivers. A full development of such optimisation is beyond the scope of this chapter.

The QoS analysis of such randomised gossip protocols seen in the literature may not be sufficient for our purposes of building QoS adaptive subsystems. For example, the

evaluation of latency distribution of [2] is done in terms of gossip rounds, rather than in global time. This is also true of randomised protocols that solve consensus (e.g., [Ben-or, RandomPE]): the probability of termination is shown to approach 1 as the number of rounds increase. The QoS analysis will additionally require the distribution of the time taken to achieve consensus, given the delay distribution and loss probabilities.

## 7. Concluding Remarks

Approaches to distributed system designs have thus far assumed two broad classes of computational and communication models: in the synchronous model, processing and communication delays are considered to be uniformly distributed in a known range; in the asynchronous model on the other hand, delays are finite but without any assumption on the ability to deduce delay bounds or delay distribution. So, any bound on delays, deduced however judiciously, is subject to being violated.

Timed asynchronous (TA) model of Fetzer and Cristian [9] and the overlay approach of Verissimo and Casimoro [19] are two well-known attempts at providing useful service properties when the system cannot be regarded to be synchronous. In TA model, the user can specify a likely bound on communication delays, and excessively delayed messages are observed (using a fail-aware datagram service) as failures and a suite of middleware protocols have been developed which guarantee correct services if the bound holds but can lack liveness on occasions when the bound does not hold. The overlay approach effectively combines the synchronous and asynchronous models to assure service liveness.

We have introduced a system model called the *probabilistic asynchronous* model (PA model) which characterises the context in which many practical and the Internet-based applications are built. Specifically, the model regards that basic services and system components (e.g., network services) offer Quality of Service (QoS) guarantees that are *probabilistic* in nature, admitting that the guarantees may not be met on odd occasions within the remit of the stated probabilities. The model provides a basis for designing and reasoning about systems that adaptively meet QoS obligations to end users; eventual correctness is never compromised when components fail in their QoS obligations.

The classical synchronous model is a special case of the PA model, with an implication that any problem that is unsolvable in the former is also not solvable in the PA model. We have also noted that the PA model with assumptions on message loss probability  $q$  and finite support distributions becomes a special case of the asynchronous model. With this insight, we have shown how PA protocols can be derived (where feasible) from existing synchronous and asynchronous ones.

## Acknowledgement

This work is part-funded by the UK EPSRC under grant GR/S02082/01 (Protocols for Programmable Networks) and the European Union Project IST-2001-34069: TAPAS (Trusted and QoS Aware Provision of Application Services). Discussions with, and clarification provided by, Isi Mitrani were extremely useful.

## References

- [1] Arvind K. Probabilistic Clock Synchronisation in Distributed Systems. *IEEE Transactions in Parallel and Distributed Systems*, 5(5):475-487, May 1994.
- [2] Birman K. et. al. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2): 41-88, May 1999.
- [3] Birman K and Joseph T. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1): 47-76, February, 1987.

- [4] Bracha G. and Toueg S. Asynchronous consensus and Broadcast Protocols, in the *Journal of the ACM*, 32: 824 – 840, Oct. 1985.
- [5] Chandra T D and Toueg. Unreliable Failure Detectors for Reliable Distributed Systems, *Journal of the ACM*, 43(2): 225-267, 1996.
- [6] Chandra T D, Hadzilacos V and Toueg. The weakest Failure Detector for Solving Consensus, *Journal of the ACM*, 43(4), pp. 685 - 722, 1996.
- [7] Chen W, Toueg S and Aguilera M K. On the Quality of Service of Failure Detectors, *IEEE Transactions on Computers*, 51: 561-580, May 2002.
- [8] Cristian F. Probabilistic Clock Synchronisation. *Distributed Computing*, 3(3):146-158, 1989.
- [9] Cristian F and Fetzer C. The Timed Asynchronous Distributed System Model, In *IEEE Transactions on Parallel and Distributed Systems*, 10 (6): 642-57, June 1999.
- [10] Ezhilchelvan P D, Mostefaoui A and Raynal M. Randomized Multivalued Consensus. *In the proceedings of the fourth International IEEE Symposium on Object oriented Real-time Computing (ISORC)*, pp. 195-201 May 2-4, 2001.
- [11] Ezhilchelvan P D and Shrivastava S K. *rel/REL*: A Family of Reliable Multicast Protocols for Distributed Systems, *Distributed Systems Engineering*, 6: 323 – 331, 1994.
- [12] Floyd S. et. al. A reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. *SIGCOMM Computer Communications Review*, 25(4): 342-356, October 1995.
- [13] Guerraoui R. Revisiting the relationship between Non-blocking Atomic Commitment and Consensus. *In Proceedings of the Ninth International Workshop on Distributed Algorithms*, Springer-Verlag, September 1995.
- [14] Gupta I, Birman K and Van Renesse R. Fighting Fire with Fire: Using a Randomised Gossip to Combat Stochastic Scalability Limits. *Quality and Reliability Engineering International* 18: 165-184, 2002.
- [15] Hadzilacos V. and Toueg S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, (Ed.) S Mullender, Addison-Wesley, 1993, pp. 97-146.
- [16] Kopetz H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997, ISBN 0-7923-9894-7.
- [17] Park K and Willinger W. *Self-Similar Network Traffic and Performance Evaluation*. John Wiley & Sons, 2000, ISBN 0-471-31974-0.
- [18] Pease M., Shostak R. and Lamport L. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228-234, April 1980.
- [19] Verissimo P and Casimiro A, The Timely Computing Base Model and Architecture, *IEEE Transaction on Computing Systems*, 51(8): 916-930, 2002.