Timeout-based Adaptive Consensus: Improving Performance through Adaptation*

Mônica Dixit, Henrique Moniz, António Casimiro
Faculty of Sciences, University of Lisbon
{mdixit,hmoniz,casim}@di.fc.ul.pt

ABSTRACT

Algorithms for solving distributed systems problems often use timeouts as a means to achieve progress. They are designed in a way that safety is always preserved despite timeouts being too small or too large. A conservatively large static timeout value is usually selected, such that the overall system performance is acceptable in the normal case. This approach is good enough in stable environments, but it may compromise performance in more dynamic settings, such as in wireless networks. In this case, a better approach is to dynamically adjusting timeouts according to the observed network conditions.

This paper clearly illustrates the achievable improvements and thus justifies the importance of using adaptive protocols in dynamic environments. We describe our pragmatic approach to transform a static timeout-based consensus protocol for ad hoc wireless networks into a fully autonomic and adaptive solution. Our comparative experiments, performed in a wireless environment, show that in contrast with the original static protocol, the adaptive solution leads to an almost constant bandwidth utilization despite increasing the number of consensus participants, and the overall consensus execution time increases linearly instead of exponentially.

Categories and Subject Descriptors

C.0 [**System Architectures**]; C.4 [**Performance of Systems**]

General Terms

Design, Reliability

Keywords

Consensus, adaptation, probabilistic monitoring, performance

*This work was partially supported by the EC, through project IST-FP7-257475 (MASSIF), and by FCT, through project TRONE (CMU-PT/RNQ/0015/2009) and the Multiannual and CMU-Portugal programmes.

SAC'12 March 22–29, 2012, Riva del Garda (Trento), Italy

1. INTRODUCTION

The consensus problem is a fundamental building block in the design of distributed systems, as it contributes to the coordination of actions in order to achieve consistent decisions. The solution for many agreement problems, such as atomic broadcast or leader election, relies on the ability to achieve some form of consensus among a set of processes.

The well-known FLP impossibility result [8] states that the consensus problem cannot be solved in asynchronous systems subject to process failures. The typical solution to circumvent this impossibility is to strengthen the timing assumptions of the system, either implicitly through a failure detector or explicitly through partial synchrony models, so that processes can resort to some timing information for making progress. This usually involves setting a timeout and, if the timeout expires, take appropriate measures (e.g., marking some process as faulty), instead of waiting indefinitely for messages that may never arrive.

Timing information, however, is inherently unreliable in asynchronous systems. Therefore, the design of distributed consensus protocols is usually centered at preserving safety regardless of the underlying timing behavior, while liveness is achieved on an eventual basis, when the system exhibits some minimum level of synchrony.This design principle, as sound as it is, relegates the problem of selecting an appropriate timeout to a secondary plane, because its value has no effect on the safety of the protocol (and liveness just requires for this value to eventually become sufficiently large). Thus, this problem is often dismissed as being 'merely' an engineering decision, where a fixed timeout is conservatively selected based on ad hoc approaches or on empirical observations of the network. However, while in any soundly designed protocol correctness is not dependent on specific timeout values, performance is. A too small timeout will raise many false positives (if failure detection is involved) or cause too much contention (if retransmission is used). A too large timeout will hinder a quick recovery from failures.

Therefore, to reason about performance, a crucial issue is the characterization of the temporal behavior of the network on which the consensus protocol is executed. And this is particularly important when considering wireless environments, in which network delays are strongly exposed to the effects of contention [10]. If a correct timeout can be selected, and consensus can be dynamically adapted, then performance can be improved.

In this paper we describe a pragmatic approach to build and tune an adaptive consensus protocol for best performance. The presented solution performs well despite

changes in network conditions and without sacrificing safety, thus clearly illustrating the importance and usefulness of adaptation and autonomic behavior. The concrete performance improvements are measured in terms of the consensus latency and network utilization. We compare the performances of the adaptive and of the static consensus protocols in order to quantify the benefits of dynamic timeout adaptation. In particular, we study the impact of a varying number of participating processes on the observed performance. While the static protocol version exhibits a significant performance degradation even with a small increase in the number of processes, in the dynamic version the network load generated by each process remains essentially constant, and the execution time only increases linearly, instead of exponentially.

Our solution uses *Adaptare* [7] to drive the adaptation process, which is a framework for dependable adaptation that employs probability techniques for monitoring and predicting future values of system variables. Using *Adaptare* as a timeout provisioning service has two main advantages: (i) we have confidence on the provided timeouts, because they are computed to ensure that they will hold with a *known probability*, specified by the user; (ii) since *Adaptare* is an independent service, it is easier to apply the proposed approach to transform other static protocols into dynamic ones. In this sense, our methodology is generally applicable and this paper contributes with the basic principles and guidelines that must be followed for that purpose.

The paper is organized as follows. The next section discusses some related work. In Section 3 we briefly describe the consensus protocol studied in this paper. Section 4 illustrates the impact of network load on the consensus execution time. In Section 5 we describe our approach to transform the static timeout-based consensus protocol into an adaptive solution, with the objective of improving performance. In Section 6 we present our experimental evaluation, which compares the performance of the static and adaptive versions of this protocol in a wireless ad hoc network. Finally, Section 7 concludes the paper.

2. RELATED WORK

The problem of dynamically selecting timeouts has been studied in several different contexts. For example, this is a critical problem for real-time multimedia systems and applications, because in those systems the delay in detecting errors caused by conservative large timeout would compromise the quality of service perceived by end users. There are many works that propose solutions for timeout selection and adaptation in this context, which rely on the ability to monitor several operational parameters (e.g., [2]).

The research area of networks and distributed systems is also concerned with adaptation and timeout selection, especially with the growth of complex dynamic systems, based on wireless networks, and mobile applications, in which making static timing assumptions may compromise the system performance, or even the safety of protocols and algorithms.

In the networks field, a well-known approach for timeout selection is the retransmission timeout estimation used by the TCP protocol. It implements a very simple algorithm introduced in [9], in which retransmission timeouts are computed based on the observed round-trip times and their variations. However, some researchers have shown that the TCP congestion control mechanism is not appropriate for wireless networks, given that common wireless errors are mistaken for congestion, causing a reduction in the transmission rate and compromising the network throughput. In general, the proposed solutions for this problem are based on extensions to the link layer protocols [13, 16].

Timeout-based adaptive solutions have also been proposed in the context of failure detection, with the objective of improving the tradeoff between detection time and accuracy of failure detectors. Typically, timeouts are derived from some statistics applied to a number of observed heartbeats delays (e.g. average delay), and on different mechanisms to compute safety margins [3, 12]. Other solutions focus on adapting parameters for failure detection according to specified QoS requirements. In particular, the failure detector proposed in [5] follows an approach based on the feedback control theory to compute the interrogation period, adapting both timeout and period in runtime. In [6] we propose an adaptive failure detector, also using *Adaptare* as a timeout provisioning service (as we do in this work), and compare its performance with other adaptive failure detectors, including the QoS-driven approach presented in [4].

In general, the works mentioned above handle the problem of timeout selection in specific application contexts, without separating the monitoring and adaptation aspects from the application semantics. Such tailored approaches can deliver optimized results, but make it difficult to reuse the monitoring and adaptation mechanisms in other contexts.

In contrast with the typical approaches, our solution has two fundamental advantages. First, we propose to use a framework for monitoring and adaptation support that may be used as a service and, in that sense, is fully independent of the specific application context. This is why it becomes easier to transform static timeout-based protocols into dynamic ones. Second, the framework is driven by a dependability requirement expressed though a coverage value. This is the crucial parameter to be set, and defines the minimum expected probability that the computed timeout will be large enough for the application purposes. The coverage value does not depend on the execution environment, thus no fine tuning process is required to reuse adaptive solutions that follow our approach in different environments: they automatically adapt to new conditions and networks, computing the necessary timeouts to secure the expected coverage.

3. CONSENSUS PROTOCOL

The consensus protocol introduced in [11] solves the k-consensus problem in wireless ad hoc networks. In the k-consensus problem, each process p_i proposes an initial binary value $v_i \in \{0, 1\}$, and at least $k > \frac{n}{2}$ of them have to agree on a common proposed value, where n is the number of participant processes. The remaining processes are not required to decide, but if they decide, it must be on the same value decided by the k processes.

The work assumes a communication failure model [14], which is more appropriate to represent wireless ad hoc networks, but an impossibility result states that under this model there is no deterministic algorithm that allows n processes to reach k-agreement, if more than $n-2$ transmission failures occur in a communication step [14]. The protocol introduced in [11] circumvents this impossibility by employing randomization to tolerate omission transmission faults.

The protocol executes in rounds. In a round, each process p_i broadcasts a message containing its identifier, pro-

posal value and other variables comprising its internal state. Then it waits for messages broadcast by the other processes. When messages from the majority of the processes are received, process p_i will make progress by analyzing them, updating its state and possibly deciding on some value or initiating a new round. However, as assumed in the communication failure model, some messages that a process is supposed to receive may be lost. This may delay p_i (since it will need to wait for slower messages) or may even prevent progress to be done (if too many messages are lost). Therefore, in each round, process p_i waits for messages only during a pre-defined timeout. If not enough messages are received within this timeout, its state does not change and p_i starts a new round by retransmitting the original message. This protocol ensures safety regardless of the number of omission faults in each round, while liveness is guaranteed in rounds where the number of omission faults is $f \leq \lceil \frac{n}{2} \rceil (n-k)+k-2$.

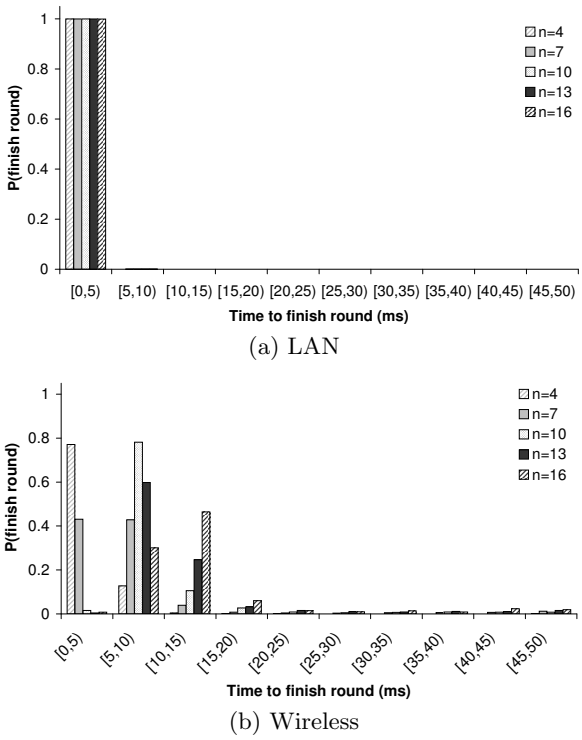
4. IMPACT OF NETWORK CONDITIONS

In this section we show that the performance of a distributed protocol can be seriously affected by varying network conditions, in particular if wireless environments are considered. This motivates the need for practical solutions to deal with such variations, based on adapting temporal variables like timeouts, aiming at achieving the best possible performance under the available conditions.

We performed practical experiments that compare the execution time of a round of the considered consensus protocol in a local area network (LAN) and in a wireless ad hoc network. The LAN experiments were performed using a cluster in our laboratory, while the Emulab testbed [15] was used for wireless experiments. We measured the duration of each round for varying load conditions. In fact, the load was controlled and dictated by the number of processes participating in the consensus execution (set to 4, 7, 10, 13, and 16). As we wanted to analyze the impact of increased load in the round execution time, the static timeout was set to a very large value (1 second). This provides enough time for all necessary messages to be received and thus gives a precise view of the round duration, even if it takes a long time. Exceptional cases of rounds in which half or more messages were lost, blocking the protocol and causing timeout expiration, were discarded.

The histograms presented in Figure 1 show the distribution of round execution times in the two networking environments for the different load conditions corresponding to varying number of processes. The LAN environment is stable in the sense that a small variation in the number of processes does not affect the execution time of a round significantly (Figure 1(a)). Rounds are always completed in less than 5ms, allowing the original version of the consensus algorithm, which uses a fixed timeout of 10ms, to perform well in this network. In fact, we observed that a timeout of 600μs would be sufficiently large to complete a round in these very stable conditions.

On the other hand, the execution of consensus rounds in the wireless network (Figure 1(b)) is clearly slower, with latencies of more than one order of magnitude higher than in LANs, and the number of participating processes becomes more relevant. While a timeout of 10ms appears to be sufficient for consensus among up to 10 processes, for 13 or 16 processes the timeout should be of at least 15ms to prevent unnecessary retransmissions. It is easy to conclude that in



(a) LAN

(b) Wireless

Figure 1: Comparing round execution time in LAN and wireless networks.

the wireless setting the selection of the appropriate timeout is a crucial issue, which would not be the case in LAN environments.

Note that in these experiments the consensus protocol was implemented over UDP, which means that the transport layer does not employ any kind of retransmission mechanism or congestion control. This is in accordance with the assumed communication failure model, and implies that retransmissions (if necessary) must be handled by the protocol. It is thus the responsibility of the protocol to select the appropriate timeout values. These must be small enough to quickly initiate a new round instead of waiting for slow (e.g. due to backoff or waiting delays of the 802.11 MAC layer) or lost messages, but large enough to avoid unnecessary retransmissions and consequent congestion, which would ultimately slow down the protocol. Moreover, these values should be adapted according to the observed conditions, that is, the protocol should be adaptive.

5. ACHIEVING ADAPTIVE CONSENSUS

The consensus protocol described in Section 3 relies on a timeout to decide whether messages should be retransmitted and when that should be done. In this section, we describe our approach to transform it into an adaptive protocol. Our goal is to improve its performance by dynamically adjusting the timeouts in response to variations on the network delays.

The architecture of our solution is very simple: each consensus process has its own instance of the timeout provisioning service (*Adaptare*), so that timeout selection decisions are fully decentralized. Furthermore, there is a logical separation between the timeout-based consensus protocol and timeout calculation functions, which makes it easier to ap-

ply the approach without the need of significant changes on protocol code, as detailed ahead.

5.1 Adaptare: timeout provisioning service

We propose the use of *Adaptare* to compute timeouts during the protocol execution [7]. *Adaptare* is a framework developed to support adaptive systems in stochastic environments, driving the adaptation process according to dependability requirements specified by the client application. For self-containment, and since it is important to understand the interface provided by *Adaptare* that is used by the adaptive consensus protocol, we provide a short description of *Adaptare* in what follows. The interested reader is referred to [7], which provides a complete description of *Adaptare*.

Figure 2 illustrates *Adaptare*'s operation. Three parameters must be provided at the framework interface: (i) recent samples of the monitored temporal variable; (ii) the history size h, which defines the number of samples that will be analyzed by the framework; and (iii) the required coverage for the computed bound, which is the probability that this bound will hold given the network conditions. Then, *Adaptare* performs a probabilistic analysis of the samples in the history, in order to determine an upper bound on the observed variable, according to the required coverage.

Probabilistic characterization. The framework is based on the assumption that the system (represented by the monitored variable) behaves stochastically, alternating periods during which the environment conditions remain fixed (*stable phases*), with periods during which those conditions change (*transient phases*). During stable phases, the statistical process that generates the data flow may be characterized, and hence we can compute the corresponding distribution using an appropriate number of samples. On the other hand, if the environment conditions are changing, then the associated statistical process is actually varying and no fixed distribution can describe its real behavior.

Adaptare employs two goodness-of-fit (GoF) tests for the analysis of input samples in order to determine whether the system is in a stable period and, if this is true, which probability distribution better describes the current state: the Kolmogorov-Smirnov (KS) test and the Anderson-Darling (AD) test [1]. Five distributions are verified: exponential, shifted exponential, Pareto, Weibull, and uniform. The parameters for those distributions are estimated from the samples, using the maximum likelihood estimation (MLE) and the linear regression methods.

Bound computation. Once the environment characterization is completed, *Adaptare* is able to compute a new dependable upper bound for the observed variable. For stable phases, this bound is derived from the cumulative distribution function (CDF) of the identified distribution. Formally, the CDF represents the probability that the random variable X takes on a value less than or equal to t ($F_X(t) = P(X \leq t)$). This definition allows to directly use the specified coverage parameter, which is, in fact, the probability that the variable value will be lower than the computed timeout. Thus, t is the new timeout (output bound), obtained by directly applying the required coverage and the estimated parameters into the distribution's CDF.

Whenever it is not possible to detect a distribution from the input samples (transient periods), *Adaptare* computes a conservative bound, based on the one-sided inequality of probability theory, which holds for all distributions.

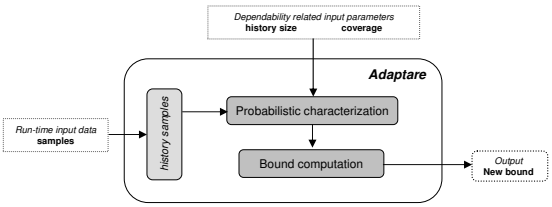


Figure 2: *Adaptare* framework

One important practical property of the framework is the separation between monitoring and characterization. The client application may feed *Adaptare* with new observations of the variable of interest at any time during the system execution. The h (history size) most recent values constitute *Adaptare*'s internal history, with h provided at the framework interface. The characterization process is triggered only when a new timeout is requested: at this point, *Adaptare* runs the probabilistic mechanisms over the history of samples and returns the new timeout to the application.

5.2 Protocol instrumentation

Typically, timeout-based protocols rely on a fixed timeout value that defines an upper bound on the time that a process should wait for some event. In order to make timeouts adaptive using *Adaptare*, it is necessary to: (i) identify the system variable that should be bounded by the timeout; (ii) instrument the protocol in order to collect samples of this variable that will be fed into *Adaptare*, and (iii) search for places in the protocol where the timeout is used, adding a call to *Adaptare* in order to update the timeout.

In the consensus protocol considered in this paper, the timeout defines how long a process will wait for the reception of messages from the majority of the processes, after initiating a round. Therefore, the variable to be monitored is the time elapsed between the beginning of a round and the reception of each message for that round. Even those messages that arrive after the round terminates should be considered, otherwise we would have information only about timely messages, compromising *Adaptare*'s analysis.

Given that, we instrumented the consensus protocol to keep track of the time at which each round is initiated. When a new message is received, the algorithm determines the round to which the message belongs, calculates the amount of time elapsed since the beginning of that round, and sends this value to *Adaptare*. Besides that, we only had to modify the consensus code so that, before a message is broadcast, a request is made to *Adaptare* to obtain the timeout value that should be used in that round.

5.3 Configuring Adaptare

One fundamental issue in the proposed approach is the correct configuration of *Adaptare*, so that dynamically obtained timeout values are appropriate to achieve the desired performance. In essence, when developing the adaptive solution it is necessary to understand which are the performance objectives of the application or protocol, and translate them into a coverage requirement, provided to *Adaptare*.

In this case, the reasoning was the following. During the consensus execution, all processes will be sending and receiving messages, and will be able to make progress as long as they receive enough messages during a defined time interval after they initiate a round (by sending a message). They do

not need to receive all the possible messages. In fact, they only need to receive messages from the majority of the processes, that is, more than $\frac{n}{2}$. Given that the total number of messages they could receive in a round is n (one from each process, including its own), the fraction of required timely messages is thus $\frac{\lfloor \frac{n}{2} \rfloor + 1}{n}$. In other words, we can say that the selected timeout must be such that it allows messages to be timely with a minimum coverage given by $C = \frac{\lfloor \frac{n}{2} \rfloor + 1}{n}$. This defines the coverage value that must be used in the configuration of *Adaptare*, which will yield a timeout value that will be sufficiently large to allow progress to be made for the observed environment conditions.

Note that the timeout might not be always sufficiently large, leading to unnecessary retransmissions. This is natural when considering stochastic processes, with continuously changing environments. For instance, if at a certain moment there is an increase in the observed delays, this may lead, in a first moment, to retransmissions caused premature timeouts. As a reaction, these increased delays will be fed into *Adaptare*, which will also provide increased timeout values to the protocol. This automatic adjustment will bring the protocol back to the expected good behavior, in which it will wait just the necessary amount of time for making progress.

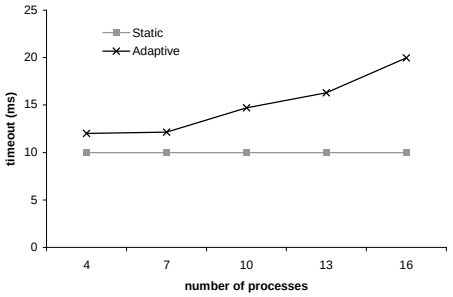
6. PERFORMANCE EVALUATION

We executed a set of experiments in order to quantify the improvements that are achieved by our adaptive consensus protocol. We compared the performance of the static version presented in [11] and of the adaptive version. The experiments were carried out on the Emulab testbed [15]. A total of 16 nodes were used, each one with the following characteristics: Pentium III processor, 600 MHz of clock speed, 256 MB of RAM, and 802.11 a/b/g D-Link DWL-AG530 WLAN interface card. The operating system was the Fedora Core 4 Linux with kernel version 2.6.18.6. The nodes were located on the same physical cluster and were, at most, a few meters distant from each other. Since the Emulab environment is not isolated, and our experiments could suffer from the interference of other nodes outside our control, we executed the experiments in six different days, to mitigate possible occasional interferences on the global results.

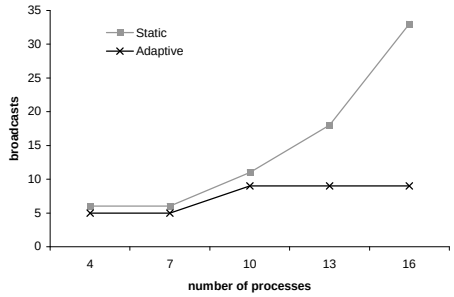
The number of processes participating in the consensus execution was set to 4, 7, 10, 13, and 16. An experiment comprises 20 consecutive executions of the static and the adaptive algorithms for a given n. We executed five experiments per day (one for each value of n), during the six different days, leading to a total of 1200 consensus executions.

The static version of the protocol was configured to use a timeout of 10ms. This value was obtained by running a set of empirical tests with different static values, being the value that provided the better results on average. The initial timeout for the adaptive version is also 10ms, but it is dynamically adjusted according to *Adaptare* outputs. *Adaptare* was configured to use a history size of $h = 30$, as recommended in [7]. The required coverage was set to $C = \frac{\lfloor \frac{n}{2} \rfloor + 1}{n}$, as explained in Section 5.3.

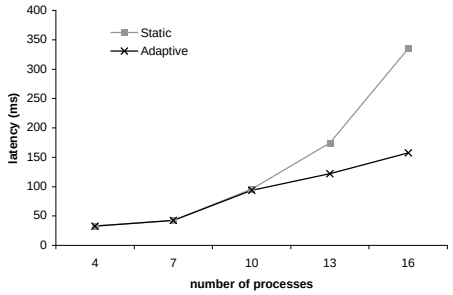
Figure 3(a) shows the average timeouts. The lower timeout is the fixed value of the static algorithm, 10ms. The adaptive algorithm presents average timeouts from 12ms to 20ms, depending on the number of processes participating in the consensus execution. Since every process in the adaptive



(a) Average timeout



(b) Average number of broadcasts per process



(c) Average latency

Figure 3: Adaptive vs. static consensus

consensus computes its own timeouts independently, the local average differs from the global one. We note that this difference was, at most, 3.5ms.

Analyzing the number of broadcast messages sent by each process (Figure 3(b)), the relation between timeouts and retransmissions is evident. In the adaptive consensus, which used higher timeouts, the number of broadcasts per process was significantly lower. This is a clear indication of the benefits that may be achieved by adapting the timeout. When consensus is executed by a higher number of processes, creating more contention and increased transmission delays, the static version uses an inadequate timeout value, whereas the adaptive version automatically adjusts the timeout to fit the environment conditions. By increasing the timeout, the adaptive version avoids premature retransmissions and hence prevents the network load to increase even more and affect negatively the observed network delays, as well as the consensus latency.

Increasing the timeout also increases the delay of necessary retransmissions. Ultimately, this delay could impact the consensus execution time, which is the fundamental performance indicator perceived by end users. However, our

results show that the overall execution time is nevertheless better for the adaptive version, which indicates a positive trade-off in favor of the increased timeouts. In fact, Figure 3(c) shows the latency improvements for the dynamic version, which were particularly visible for the scenarios with 13 and 16 processes. Therefore, this also confirms the experiments in the wireless environment presented in Section 4, which suggested that a timeout of 10ms would be sufficient for consensus among up to 10 processes, but too small for more processes. Both static and adaptive versions of the protocol achieved similar latencies in the scenarios with 4, 7 and 10 processes. However, in executions with 13 and 16 processes, with the timeout increasing to 16ms and 20ms (in average), latency improvements were about 30% and 53%, respectively.

7. CONCLUSIONS

In this paper we show that the run-time performance of distributed algorithms can be improved by adapting timing variables and timeouts used in algorithms to the actual conditions of the environment. The problem is particularly relevant in dynamic environments, specially when communication latencies can be affected by a varying number of processes and dynamic network load. Therefore, we focus on the practical aspects of implementing adaptive distributed protocols to operate in dynamic networks. Following a pragmatic methodology, we describe a simple approach to transform a static timeout-based consensus protocol for wireless ad hoc networks into an adaptive protocol.

A fundamental building block of our solution is the adaptation support framework, which continuously provides the timeout that must be used during the protocol's execution. We use *Adaptare*, a generic framework to support adaptation, which is easily configured and may be used as an independent building block, in the context of different applications.

The results of the experimental evaluation attest the benefits of using adaptive protocols in wireless networks. While the static version of the consensus algorithm can perform well in just one scenario, the one for which the static timeout has been determined, the adaptive version is able to adapt to any timeliness conditions.

As future work, we point out that our evaluation could be extended to consider even more dynamic scenarios, with varied distances and mobility between nodes, which was not possible to control in the Emulab testbed. Another open issue that could be investigated is how to automatically compute *Adaptare*'s coverage, facilitating the design of adaptive versions of static protocols in which performance is dictated by different parameters and observed variables.

We believe that our contribution is important both from an engineering and from a practical perspective. By using an well-defined service for timeout provisioning we show that it is possible to easily add adaptive behavior to a previously static protocol. The practical outcome is that it becomes possible to improve performance without sacrificing other properties, namely safety. This is particularly important, as shown, in load-sensitive wireless environments.

8. REFERENCES

[1] A. O. Allen. *Probability, statistics, and queueing theory with computer science applications*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[2] A. C. Begen and Y. Altunbasak. An adaptive media-aware retransmission timeout estimation method for low-delay packet video. *IEEE Transactions on Multimedia*, 9(2):332–347, 2007.

[3] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *32nd IEEE Int. Conf. on Dependable Systems and Networks*, pages 354–363, 2002.

[4] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002.

[5] A. S. de Sá and R. J. de Araújo Macêdo. QoS self-configuring failure detectors for distributed systems. In *10th Int. Conf. on Distributed Applications and Interoperable Systems*, pages 126–140, 2010.

[6] M. Dixit. *Support for Dependable and Adaptive Distributed Systems and Applications*. PhD thesis, Department of Informatics, University of Lisbon, 2011. http://hdl.handle.net/10455/6749.

[7] M. Dixit, A. Casimiro, P. Lollini, A. Bondavalli, and P. Verissimo. Adaptare: Supporting automatic and dependable adaptation in dynamic environments. *ACM Transactions on Autonomous and Adaptive Systems*, (to appear). Also as Dep. of Informatics, Univ. of Lisbon, Technical report TR-09-19, 2011.

[8] 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, 1985.

[9] V. Jacobson. Congestion avoidance and control. *SIGCOMM Computers Communication Review*, 18:314–329, Aug. 1988.

[10] A. P. Jardosh, K. N. Ramachandran, K. C. Almeroth, and E. M. Belding-Royer. Understanding congestion in IEEE 802.11b wireless networks. In *5th ACM SIGCOMM Conf. on Internet Measurement*, pages 25–25, 2005.

[11] H. Moniz, N. F. Neves, M. Correia, and P. Verissimo. Randomization can be a healer: consensus with dynamic omission failures. In *23rd Int. Conf. on Distributed Computing*, pages 63–77, 2009.

[12] R. C. Nunes and I. Jansch-Porto. QoS of timeout-based self-tuned failure detectors: The effects of the communication delay predictor and the safety margin. In *34th IEEE Int. Conf. on Dependable Systems and Networks*, pages 753–761, June 2004.

[13] I. Psaras, V. Tsaoussidis, and L. Mamatas. CA-RTO: A contention-adaptive retransmission timeout. In *14th Int. Conf. on Computer Communications and Networks*, pages 179–184, 2005.

[14] N. Santoro and P. Widmayer. Time is not a healer. In *6th Symposium on Theoretical Aspects of Computer Science*, pages 304–313, 1989.

[15] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *5th Symposium on Operating Systems Design and Implementation*, pages 255–270, 2002.

[16] G. Xylomenos and C. Tsilopoulos. Adaptive timeout policies for wireless links. In *20th Int. Conf. on Advanced Information Networking and Applications*, pages 497–502, 2006.