Phoenix

A Toolkit for Building Fault-Tolerant, Distributed Applications in Large Scale*

Christoph Malloth, Pascal Felber, Anrdé Schiper, Uwe Wilhelm
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH - 1015 Lausannne (Switzerland)
phoenix@lse.epfl.ch

July 18, 1995

Abstract

Large scale systems are becoming more and more common today. There are many distributed applications emerging that use the capability of world-wide internetworking. Since a lot of applications need insurance of consistency even in the presence of failures, an adequate support for fault-tolerance is necessary. This can be provided by different paradigms and their implementations. Unfortunately, most of these implementations aim only local area networks, whereas our system, called Phoenix, will aim large scale where additional failure types have to be overcome. This paper shows the problems due to large scale, the limits of actual implementations, and our proposition to solve them.

1 Introduction

Large scale distributed systems are in wide use today. Internet is the prime example for a large scale network, interconnecting millions of computers. Many applications already use this connectivity, e.g. a large scale cooperative editing system where partners of a project can be on different continents, or an air traffic control system which is in charge of more than only a local sector. Unfortunately, applications often lack support for fault-tolerance. There are systems which tolerate failures, but this fault-tolerance is limited to site failures. Considering large scale, link failures are not unusual to occur, and the systems should deal with this new kind of failure.

We consider here the replication of a service as an adequate method to provide fault-tolerance. In order to guarantee a consistent state among the replicated servers in spite of failures, the *virtually synchronous communication* paradigm can be applied [3]. Isis was the first programming toolkit to implement this paradigm, other systems followed later, among them Horus [11] or Transis [1]; these toolkits, however, are still destined to local area networks where link failures are uncommon to occur. Phoenix, a programming toolkit developed in our laboratory, will provide a new implementation of the *virtually synchronous communication* paradigm suitable for large scale [7] as it is considering link failures as well. In addition, Phoenix provides some communication primitives not present in other platforms. These primitives provide a clear message semantics considering correct and incorrect processes.

Section 2 gives a brief description of our view of the *virtually synchronous communication* paradigm and the new failure types, due to large scale, to which it is exposed. Section 3 discusses the differences between Phoenix and other toolkits, especially their concerns about large scale. Section 4 concludes the paper with some final remarks.

*Research supported by the "Fonds national suisse" and OFES under contract number 21-32210.91, as part of the ESPRIT Basic Research Project BROADCAST (number 6360)

2 View synchronous communication and large scale

2.1 The paradigm

View synchronous communication[1] can be defined on a group of processes. Given a group g, a view is a set of correct processes as perceived e.g. by a membership service. We note $\mathsf{View}_k(g)$ as the kth view of group g. The view of group g evolves as (1) processes in g crash[2], or (2) processes join or leave g. Given a group g, *view synchronous communication* is defined informally by the following two properties: (1) **Unique view sequence:** processes in g agree on the sequence of views $\mathsf{View}_0(g)$, $\mathsf{View}_1(g)$, ..., $\mathsf{View}_k(g)$, ..., and (2) **Agreement on message delivery:** processes in g agree on the set of messages delivered between the delivery of each pair of view changes, i.e. they agree on the set of messages delivered between the delivery of $\mathsf{View}_k(g)$ and the delivery of $\mathsf{View}_{k+1}(g)$.

2.2 Partitions

A major failure type in large scale are partitions due to link failures. Whereas they exist also in local area networks, in large scale, they can partition a network in more than two separate subsets. Having more than two subsets leads to the problem, that in neither of these subsets a primary partition can be defined, a primary partition necessary to guarantee the unique sequence of views (cf. 2.1). So, what should the system do if there is temporarily no primary partition to define a new view? As the partitions are often transient, the system can wait until a primary partition and thus a new view can be defined. This implies that the failure suspicion of other hosts must be of unstable character [10], i.e. if a process suspects another, it should be able to reconsider the process alive, if it was only partitioned and actually up and running. If the failure suspicion model is of the stable type, i.e. a process considered failed, will be it forever, this type of partition will not be tolerated and the system will block in each partition. If the system does not block in minority partitions, the primary partition property can not be guaranteed and the system can have two or more partitions representing different states of the system.

2.3 Non-transitivity of communication

Another problem of large scale is the non-transitivity of communication due to simple link failures. We illustrate this problem by an example: consider three processes p_1, p_2, p_3, where the link between p_1 and p_3 has failed. p_1 actually sees only itself and p_2, the same for p_3, and only p_2 sees all three processes. As there are suspicions, the VSC protocol is launched in order to define a new view containing all *correct* processes. After the termination of the VSC protocol, the new view could be either $\mathsf{View}' = p_1, p_2$, $\mathsf{View}'' = p_2, p_3$ or $\mathsf{View}''' = p_1, p_2, p_3$, but whichever view is chosen, it leads to an instability of the system. If View' or View'' is chosen, either p_1 or p_2 is excluded from the new view and will try to join the system again, which would then trigger another view change with the same problems as in the original configuration. If View''' is chosen, then upon installation of the new view, either p_1 or p_3 would launch the VSC protocol again, as they still suspect each other.

This instability is due to the *non-transitivity* of communication between these processes and the resulting *partial* partitions [6]. A possible solution, that we call *process or site level* routing, is implemented as a layer below the VSC protocol layer. This layer provides the transitivity and eliminates the problem stated above. The idea is simple: if p_1 can not communicate with p_3, but p_1 can communicate with p_2 and p_2 with p_3, then p_1 sends all messages for p_3 to p_2, which will forward them to p_3 (for more details see [6])[3]. As a by-product, the generation of the routing tables produces at the same time the failure suspicion information for upper communication layers such as the *view synchronous communication* layer. A host is suspected as having failed if there is no more routing information to access the host. In using the routing tables for failure suspicion, false suspicions making the view of the system unstable are eliminated.

[1] We found the designation "view synchronous" more suitable than "virtually synchronous" for the given context, but they are actually the same.

[2] More precisely, when processes are suspected to have crashed, which does not exclude incorrect suspicions.

[3] A real life experience on the Internet has shown the utility of the site routing. We give just one of several concrete examples. During some testing, involving one site in Chile, one in Italy and one in Switzerland, we observed that for more than 40 minutes, IP routing was not able to send messages from Switzerland to Chile. Our routing algorithm successfully rerouted these messages through Italy.

2.4 New communication primitives

Phoenix includes new communication primitives not present in other toolkits. We give here only an informal description of those primitives, where for an exact description we refer to [8] and [12].

uniform multicast: if a process (correct or not)[4] delivers a message m, then every correct process will eventually deliver m or fail.

strong totally ordered multicast: if a process (correct or not) delivers a message m_1 before m_2 and delivers them, then every correct process will eventually deliver m_1 before m_2 or fail.

For completeness we give our informal definition of the total order multicast (actually the Isis ABCAST):

weak totally ordered multicast: if a correct[5] process delivers a message m_1 before m_2, then every correct process will deliver m_1 before m_2 or fail.

The *uniform multicast* gives actually the primitive to implement multicasts which involve stable storage.

2.5 Architecture

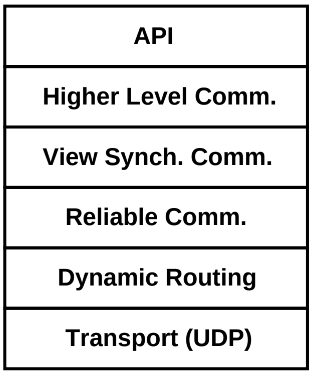

Figure 1: architecture

Figure 1 shows the actual Phoenix architecture. At the bottom layer Phoenix uses UDP as the base of communication. On top of this layer the *dynamic routing* layer provides transitive communication. The *reliable communication* layer is reponsible for retransmitting lost or corrupt messages, as well as for duplicate recognition and elimination. The *view synchronous communication* layer provides the *view synchronous multicast* and linear membership views to the upper layer, where the *higher order communication* layer implements the primitives of the previous subsection. The application programming interface is actually written in C++ and provides classes for implementing members, clients and sinks [2].

3 Discussion

In the past few years, several toolkits for fault-tolerant programming have appeared. Among them are Isis [3], Horus [11] and Transis [1], to cite only a few. These systems are all destined to local area networks and do only consider host failures and do not really consider link failures. So, in Isis, with its stable suspicion model, once a host is suspected as having failed, it is excluded from the group, even if the suspicion was due to a transient link failure and the host is actually up and running. This is exactly the point where the blocking appears due to partition. In the case of local area networks, this situation is very rare, however omnipresent in large scale. In Horus, they circumvented the blocking problem by allowing minority partitions to continue, but they lose the property of always having a primary partition [9] representing the consistent state of the group. The *view synchronous property* (cf. Section 2.1) is not ensured anymore. Phoenix is somewhere in between Isis and Horus, by using the unstable failure suspicion model, where a failed suspected host can be reconsidered alive. With this model, the system blocks for the time where no primary partition can be defined, but continues execution by the time where this primary partition can be defined. There is always a primary partition, and the state of the members in this partition is always consistent. In addition, a user can explicitly ask to continue in a minority partition, e.g. to do insecure reads, but as soon as a primary partition can be defined, the system continues with this primary partition with a consistent state. In other words, when a partition gets repaired, the state of a minority partition will be overwritten by the state of the primary partition.

[4] A process is considered as correct if it is member of the current and the next view.

[5] Notice that we consider here only correct processes whereas in the previous primitives also incorrect processes are considered.

Another problem described in the previous section, was the non-transitivity of communication. Most toolkits use UDP [4] for communication and seem to be satisfied by the corresponding dynamic routing on the network. In cases like the one described in the previous section, none of them will give a satisfying solution. Unfortunately, in large scale networks as the Internet, it is not uncommon that these cases occur. Thus, in Phoenix, an additional layer between UDP and the *view synchronous communication* layer guarantees transitivity of communication.

4 Conclusion

In this paper, a new approach to asynchronous, fault-tolerant systems is described by defining the basic properties of a group oriented, distributed system depending on a primary partition. The system, in contrast to other systems, will block only during the time where a primary partition cannot be defined. This guarantees to have always a consistent state in the system even in presence of link failures. An additional layer provides transitivity and eliminates the problem of unstable views.

The Phoenix toolkit developed around this VSC property, works in local area networks as well as in large scale. It takes advantage of the unstable failure suspicion model and the transitivity of communication in order to provide a more reliable, robust and consistent system, especially in large scale. The alternative routing scheme incurs performance loss in communication, but the gain in reliability will hopefully outweigh the loss by the provision of a system which is more stable and that have an increased availability.

We have actually an operational prototype of Phoenix, developed in our laboratory, running in a test bed involing several sites in the Internet. We have been able to sucessfully run Phoenix in large scale including sites in Europe (Switzerland, Italy) and South America (Columbia, Chile). There is already research ongoing on the successor of the actual Phoenix system, called Phoenix II [5] which will provide an adequate communication primitive to implement atomic transactions in distributed systems.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[2] Ö. Babaoğlu and A. Schiper. On Group Communication in Large-Scale Distributed Systems. In *Proceedings of the 7th ACM SIGOPS Workshop*, September 1994.

[3] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, February 1987.

[4] D. E. Comer. *Internetworking With TCP/IP: Principles, Protocols, Architecture.* Prentice Hall, Stevenage, 1988.

[5] R. Guerraoui and A. Schiper. A Gneric Multicast Primitive to Support Transactions on Replicated Objects in Distributed Systems. In *IEEE 6th Intl. Workshop on Future Trends of Distributed Systems (FTDCS-95)*, August 1995.

[6] C. Malloth. Increasing Reliability of Communication in Large Scale Distributed Systems Subject to Link Failures. In *IASTED 7th Intl. Conf. on Parallel and Distributed Computing and Systems*, October 1995.

[7] C. Malloth and A. Schiper. View Synchronous Communication in the Internet. In *Proceedings 2nd Open Workshop of the ESPRIT project BROADCAST (number 6360)*, July 1995 (also available as a Technical Report Nr. 94/84 at Ecole Polytechnique Fédérale de Lausanne (Switzerland), October 1994).

[8] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE 13th Intl. Conf. Distributed Computing Systems*, pages 561–568, May 1993.

[9] A. Schiper and A. Sandoz. Primary Partition "Virtually-Synchronous Communication" Harder than Consensus. In *8th Intl. Workshop on Distributed Algorithms proceedings (WDAG-8)*. Springer Verlag, 1994.

[10] S. Toueg T. D. Chandra. Unreliable Failure Detectors for Asynchronous Systems. In *proc. 10th annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1991.

[11] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus System. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, 1993.

[12] U. G. Wilhelm and A. Schiper. A Hierarchy of Totally Ordered Multicast. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, September 1995.