Transformations in Secure and Fault-Tolerant Distributed Computation

Felix C. Freiling[a] Neeraj Mittal[b*] Lucia Draque Penso[a†]

[a]Laboratory for Dependable Distributed Systems, RWTH Aachen, Germany
[b]The University of Texas at Dallas, USA

Abstract: We present a survey of different techniques used by the authors to transform a security or fault-tolerance problem into another with a known solution. We consider the following two cases: (1) reducing a security problem into a fault-tolerance problem; (2) reducing a fault-tolerant problem into its fault-*in*tolerant version. In these cases it is possible to reuse known solutions to construct new efficient algorithms.

1 Introduction

We investigate two distinct ways of reducing a security or fault-tolerance problem, so that it is possible to take advantage of an already existing solution. In the following, a *correct* process is one that never fails, a *non-faulty* process at a particular time (sometimes implicit) is one that has not failed yet, a *faulty* process is one that has already failed, and a *live* process is one that has not crashed yet.

2 From Fair Exchange to Consensus

The fair exchange problem is key to trading eletronic items in systems of mutually untrusted parties [AGGV05]. An algorithm that solves fair exchange must ensure that every honest party eventually either obtains its desired item or aborts the exchange (Termination). The abort option however is excluded if no parties misbehave and all items match their descriptions (Effectiveness). The algorithm should also guarantee that, if the desired item of any party does not match its description, then no party can obtain any (useful) information about any other item (Fairness).

Avoine, Gärtner, Guerraoui and Vukolic [AGGV05] investigate the fair exchange problem in a distributed system where parties are coupled with tamper-proof security modules (like

*Neeraj Mittal was supported through a travel scholarship by Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg “Software for mobile communication systems” at RWTH Aachen.

†Lucia Draque Penso was supported by Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg “Software for mobile communication systems” at RWTH Aachen.

smart cards) and no third party is available. They show that fair exchange can be solved at the party level if agreement (i.e., *consensus*) is solved at the security modules level.

The intuition is that the security modules can exchange the items and either release them to their respective parties after agreeing that everyone received the desired item, or abort after agreeing that some party misbehaved. With such an idea, it is possible to substantially simplify existing solutions for fair exchange. Moreover, the reduction opens the possibility to reuse efficient agreement protocols when implementing a solution.

3 Termination Detection: From Crash-Prone to Failure-Free

Informally, the termination detection problem [MFVP05] involves determining when a distributed computation has ceased all its activity. The distributed computation satisfies the following four properties or rules. First, a process is either *active* or *passive*. Second, a process can send a message only if it is active. Third, an active process may become passive at any time. Fourth, a passive process may become active only on receiving a message. Intuitively, an active process is involved in some local activity, whereas a passive process is idle.

In case both processes and channels are reliable, a distributed computation terminates once all processes become passive and stay passive thereafter, that is, once all processes become passive and all channels become empty. However, in a crash-prone system, once a process crashes, it ceases all its activities. Hence, any message in-transit towards a crashed process can be ignored because the message cannot initiate any new activity. Moreover, messages in-transit from the crashed process towards a live process can be either deleted or ignored as soon as the crash is detected. Therefore, a distributed computation in a crash-prone system terminates once all live processes are passive and either no channel contains a message in-transit towards a live process or no not-ignored channel contains a message in-transit towards a live process.

Mittal, Freiling, Venkatesan and Penso [MFVP05] efficiently reduce the crash-tolerant termination detection problem to the fault-intolerant case, making it possible to have a competitive crash-tolerant termination detection algorithm $\mathcal{B}$ out of a competitive fault-intolerant termination detection algorithm $\mathcal{A}$. More precisely, for both fully and arbitrary connected topologies, they show how to efficiently transform any fault-intolerant termination detection algorithm $\mathcal{A}$, that has been designed for a failure-free environment, into a crash-tolerant termination detection algorithm $\mathcal{B}$, that tolerates up to any number of process crashes without having to restart the underlying distributed application.

The main idea behind their approach is to *restart* the fault-intolerant termination detection algorithm $\mathcal{A}$, *whenever a new failure is detected* , on the set of *currently operational processes* . Note that before restarting $\mathcal{A}$, they ensure that all operational processes agree on the set of processes that have failed, so that they can guarantee that once the underlying distributed computation has terminated with respect to the set of operational processes, then it has terminated with respect to the whole set of processes. This procedure also avoids false termination announcement. Interestingly, it works whether messages in-transit from crashed processes towards a live one are deleted or ignored as soon as the crash is detected.

However, the only drawback to the restarting approach is that when $\mathcal{A}$ is restarted, a mechanism is needed to deal with any unprocessed application messages, that is, application messages that were sent before $\mathcal{A}$ is restarted but are received after $\mathcal{A}$ has been restarted. Such application messages are referred to as stale or *old application messages*. Clearly, the current instance of $\mathcal{A}$ may not be able to handle an old application message correctly. One simple tentative solution would be to hide an old application message from the current instance of $\mathcal{A}$ and deliver it directly to the underlying distributed computation. However, on receiving an old application message, if the destination process changes its state from passive to active, then, to the current instance of $\mathcal{A}$, it would appear as if the process became active spontaneously. This violates one of the four rules of the underlying distributed computation. Thus, the current instance of $\mathcal{A}$ may not work correctly in the presence of old application messages and therefore *cannot be directly* used to detect termination of the underlying distributed computation.

The situation is handled, though, by using the strategy of *superimposing* another computation on top of the underlying distributed computation. The superimposed computation is refered to as the *secondary computation* and the underlying distributed computation is refered to as the *primary computation*. As far as live processes are concerned, the secondary computation is almost identical to the primary computation except possibly in the beginning, when a process stays active with respect to the secondary computation at least until it knows that there are no more old application message to be received in the future.

In this way, whenever a process crashes and all live processes agree on the set of failed processes, *a new instance of the secondary computation is simulated* in the subsystem induced by the set of operational processes. Then, a *new instance of the fault-intolerant termination detection algorithm* is used to detect termination of the secondary computation. The older instances of the secondary computation and of the fault-intolerant termination detection algorithm are simply aborted. If the secondary computation has terminated then the primary computation has terminated as well , and if the primary computation has terminated, then the secondary computation terminates eventually.

Hence, with the help of this simple method of combining the restart of an efficient fault-intolerant solution with the wait of unprocessed in-transit messages, the authors are able to achieve an efficient fault-tolerant solution, as can be verified in [MFVP05].

Literatur

[AGGV05] Avoine, G., Gärtner, F.C., Guerraoui, R., und Vukolic, M.: Gracefully Degrading Fair Exchange with Security Modules. In: *Proceedings of the 5th European Dependable Computing Conference(EDCC)*. S. 55–71. 2005.

[MFVP05] Mittal, N., Freiling, F.C., Venkatesan, S., und Penso, L.: Efficient Reductions for Wait-Free Termination Detection in Crash-Prone Systems. Technical Report AIB-2005-12. RWTH Aachen. June 2005.