IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 3 Issue 6, June 2016
ISSN (Online) 2348 – 7968 | Impact Factor (2015) - 4.332
www.ijiset.com

Find Out the Failure Detector for Implement to DSA Operation

[1]K.Ravikumar, [2] C.Vivek
[1]Asst.professor, Dept.of.Computer science, Tamil University, Thanjavur-613010.
[2]Research Scholar, Dept.of.Computer Science, Tamil University, Thanjavur-613010.

Abstract:

The failure detector concept through two dimensions. First we study failure detectors as building blocks to simplify the design of reliable distributed algorithms. More specifically, we illustrate how failure detectors can factor out timing assumptions to detect failures in distributed agreement algorithms. Second, we study failure detectors as computability benchmarks. The protocols generated by our compiler are provably secure, in that their strength can be reduced to that of the original cryptographic Computation via simulation arguments. In particular, a failed node may corrupt its local state, send random messages, or even send Specific messages aimed at subverting the system. Many security attacks can be modeled as Byzantine failures, such as censorship, freeloading, misrouting, or data corruption.

Keywords: Asynchronous Model, Ids, Detecting

I. INTRODUCTION

In the area of concurrent computing for instance, abstractions like threads, semaphores and monitors were very helpful in understanding concurrent programs and reasoning about their correctness. In the area of distributed computation, the remote procedure call abstraction helped factor out the details of the network and was a key to the popularity of standard distributed middleware infrastructures. In short, the remote procedure call abstraction hides the possible differences between languages and operating systems on different machines, and encapsulate serialization and de-serialization mechanisms to transfer data over the wire.

This abstraction does not however help capture another fundamental characteristic of distributed systems: partial failures. Basically, if a process of some machine remotely invokes an operation on a process performing on a different machine, and the latter machine fails, an exception is raised. The way the failure is detected is usually achieved using a timeout mechanism. Typically, a timeout delay is associated with the operation and when it expires, the exception is raised.

Though quite weak, our definition of the fault detection problem still allows us to answer two specific questions: Which faults can be detected, and how much extra work from does fault detection require from the extension? To answer the first question, we show that the set of all fault instances can be divided into four non-overlapping classes,

and that the fault detection problem can be solved for exactly two of them, which we call commission faults and omission faults.

Intuitively, a commission fault exists when a node sends messages a correct node would not send, whereas an omission fault exists when a node does not send messages a correct node would send. To answer the second question, we study the message complexity of the fault detection problem, that is, the ratio between the number of messages sent by the most efficient extension and the number of messages sent by the original algorithm.

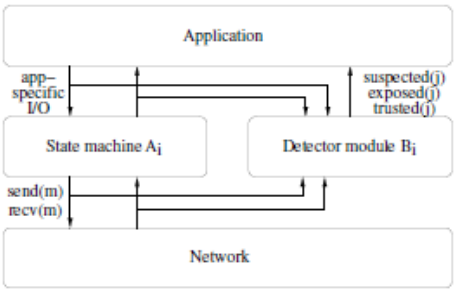
We derive tight lower bounds on the message complexity for commission and omission faults, with and without agreement. Our results show that a) the message complexity for omission faults is higher than that for commission faults, and that b) the message complexity is optimally linear in the number of nodes in the system, except when agreement is required for omission faults, in which case it is quadratic in the number of nodes.

1.1. compiler design implementation:

This section describes the design of our compiler and some implementation choices, using the DSA signature scheme as an example. Presents a high-level overview of the compiler structure. The output of the compiler the two algorithms for A and is generated from a collection of component protocols, called building blocks, and from an input file specified by the user. The component protocols correspond, intuitively, to arithmetic operations, and they are further decomposed into primitive protocols that are protocols with at most one interaction between the two parties. The input file contains the specification of a computation that will be transformed into a two-party protocol. Below, we take a bottom-up approach in detailing the compilation process.

1.2. Uses of fault detectors communication:

We consider a fully distributed detection system where every node is equipped with its own detector, which watches for faults on the other nodes. Once this detector reports a fault, the local node can respond in various ways. It can stop communicating with the faulty node. The node can then distribute the evidence in its possession to other nodes, so they can also respond and thus isolate the faulty node.



Finally, the node can initiate recovery. For example, a storage system can create additional replicas of all objects stored on the faulty node and notify a human operator, who can then repair the faulty node.

The mere presence of a detection system can reduce the likelihood of certain faults. For example, it can discourage attackers and freeloaders by creating a disincentive to cheating, since a faulty node risks isolation and expulsion from the system. Furthermore,

if the system maintains a binding from node indenters to real-world principals, then even the owner of a faulty node could be exposed and held legally responsible.

As an initial exploration into this space, however, our work so far has focused on only one simple way of combining them to reach the given input computation. The technique we have explored thus far is to compose primitive protocols into larger two-party building block protocols that implement certain operations on shared secrets. Then, our compiler emits its output using building blocks, rather than emitting instances of primitive protocols directly.

II. GREEDY TASK MAPPING

A basic greedy algorithm, which we simply call Greedy, allocates nodes to an incoming job without changing the mapping of tasks that may currently be running. It first Identifies the nodes that have sufficient available memory to run at least one task of job. For each of these nodes it computes its CPU load as the sum of the CPU needs of all the tasks currently allocated to it.

2.1. Vector Packing

The Greedy algorithm builds a solution through a succession of locally optimal decisions, but the final solution may be far from globally optimal. An alternative approach is to compute a global solution from scratch and then pre-empt or migrate tasks if necessary. As we have two resource dimensions CPU and memory, our resource allocation problem is related to a version of bin packing, known as 2D vector packing. One important difference between our problem and vector packing is that our jobs have fluid CPU needs.

2.2. New applications

New applications are driven by advances in the communication infrastructure such as the ubiquity of the Internet or the emergence of web services, coupled with increased demand for information based relationships for business or homeland security purposes. These applications often involve sensitive information related to issues such as pricing, business processes, or personal information, and their security often relies on trusting a designated trusted party such as eBay in the case of auctions.

Once such a specification is given, a compiler generates an intermediate level specification of the computation in the form of a one-pass Boolean circuit. Whereas classical theory on SFE was satisfied with the fact that it is provably possible to reduce any function to a canonical Boolean representation, we tackle for the first time actually automating the transformation, while keeping efficiency in mind.

We are planning to explore future optimizations in our compiler, such as parallelizing computation or using pre-computed tables for exponentiations with the same base. While we expect that the protocols generated by our compiler will not be as efficient as hand-tuned approaches, the performance results are already promising.

2.3. Environments

Our formulation of the fault detection problem does not require a bound on the

number of faulty nodes. However, if such a bound is known, it is possible to find solutions with a lower message complexity. To formalize this, we use the notion of an environment, which is a restriction on the fault patterns that may occur in a system. In this paper, we specifically consider environments Ef , in which the total number of faulty nodes is limited to f. If a system in environment Ef is assigned a distributed algorithm A, the only executions that can occur are those in which at most f nodes are faulty with respect to A.

III. FAILURE DETECTION OPERATION

Non-Blocking Atomic Commit. Consider the omnipresent problem of no blocking atomic commit in a distributed database. In a distributed database, data is stored at multiple sites, usually close to the location where it is used so that read and write operations on the data can be performed more efficiently. A distributed transaction groups a sequence of read and write operations together and ensures that either all are executed or none of them.

A transaction ensures that these operations are executed atomically despite site or communication failures. For simplicity, we will identify a site of the distributed database with the process of the database management system running on that site.

3.1. Asynchronous Model.

As mentioned above, having a synchronous system is not realistic in many practical situations. In fact, from an engineering perspective it makes sense to make very little assumptions about the underlying network characteristics because this achieves the highest assumption coverage. Assumption coverage refers to the probability that the assumptions about the underlying network hold in a particular mission environment. More and stronger assumptions about synchrony achieve less assumption coverage, and only a high assumption coverage ensures that the algorithms reasoning with timeouts work as expected in practice.

We are planning to explore future optimizations in our compiler, such as parallelizing computation or using pre-computed tables for exponentiations with the same base. While we expect that the protocols generated by our compiler will not be as efficient as hand-tuned approaches, the performance results are already promising.

IV. IDS LIMITED :

Intrusion detection systems (IDS) can detect a limited class of protocol violations, for example by looking for anomalies or by checking the behaviour of the system against a formal specification.

A technique that statistically monitors quorum systems and raises an alarm if the failure assumptions are about to be violated was introduced in. However, this technique cannot identify which nodes are faulty. To the best of our knowledge, were the first to explicitly focus on Byzantine fault detection. The paper also gives informal definitions of

the commission and omission faults. However, the definitions in are specific to consensus and broadcast protocols.

As an alternative to maximizing the average yield, we consider the iterative maximization of the minimum yield. At each step the minimum yield is maximized using the procedure described at the beginning of. Those jobs whose yield cannot be further improved are removed from consideration, and the minimum is further improved for the remaining jobs. This process continues until no more jobs can be improved.

V. CONCLUSIONS

A register is a shared object accessed through two operations: read and write. The write operation takes as an input parameter a specific value to be stored in the register and returns a simple indication ok that the operation has been executed. The read operation takes no parameters and returns a value according to one of the following consistency criteria. The information about crash failures needed for solving agreement, though informally anticipated earlier, were captured precisely only with the introduction Of failure detectors, and especially the notion of the weakest failure detector.

VI. REFERENCES

[1] B. Barak, A. Herzberg, D. Naor, and E. Shai. The proactive security toolkit and applications. In *Proc. 6th ACM Conf. Computer and Communications Security*, pp. 18–27, Nov. 1999.

[2] M. Bellare, S. Micali. Non-interactive oblivious transfer and applications. In *Proc. CRYPTO '89*, 1989.

[3] M. Bellare, R. Sandhu. The security of practical two-party RSA signature schemes. 2001.

[4] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc.* 1st *ACM Conf. Computer and Communications Security*, pp. 62–73, Nov. 1993.

[5] Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (Jul 1996)

[6] Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (Mar 1996)

[7] Denning, D.E.: An intrusion-detection model. IEEE Transactions on Software Engineering 13(2), 222–232 (1987)

[8] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non cryptographic fault tolerant distributed computation. In *Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC)*, pages 1–9, 1988.

[9] R. Bergamaschi, R. Damiano, A. Drumm, and L. Trevillyan. Synthesis for the '90s: Highlevel and logic synthesis techniques. In *ICCAD93 Tutorial Notes*, 1993.

[10] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of FOCS*, 2001.