$\Diamond P$-vCube: An Eventually Perfect Hierarchical Failure Detector for Asynchronous Distributed Systems

Gabriela Stein
Western Parana State University (UNIOESTE)
Cascavel, Paraná, Brazil
Gabriela.Stein1@unioeste.br

Luiz Antonio Rodrigues
Western Parana State University (UNIOESTE)
Cascavel, Paraná, Brazil
Luiz.Rodrigues@unioeste.br

Elias Procópio Duarte Jr.
Federal University of Parana (UFPR)
Curitiba, PR, Brazil
elias@inf.ufpr.br

Luciana Arantes
Sorbonne Universités, UPMC/CNRS/Inria/LIP6
Paris, France
Luciana.Arantes@lip6.fr

ABSTRACT

vCube is a virtual topology that organizes the processes of a distributed system hierarchically, presenting several logarithmic properties. Although vCube has been used as the basis to develop multiple different distributed systems abstractions, those that assume an asynchronous system deal with time uncertainty at a higher level. In this work, we present $\Diamond P$-vCube: a push-based failure detector for asynchronous distributed systems that can guarantee not only completeness but also eventual accuracy. The underlying system is assumed to be fully-connected so that any pair of processes can execute tests on each other. Since there are no limits on communication delays and process execution time, false suspicions may occur. To guarantee eventual accuracy, a process exits the system when it learns that it has been suspected by another process or suspects every other process. The proposed algorithm was implemented with simulation and compared with the traditional all-to-all solution and a ring approach. Results confirm the scalability of $\Diamond P$-vCube, which represents a compromise between how fast actual failures are detected and the cost in terms of the number of messages employed.

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**; **Fault-tolerant network topologies**.

KEYWORDS

dependability, fault-tolerance, unreliable failure detector, autonomous systems, distributed algorithms

ACM Reference Format:
Gabriela Stein, Luiz Antonio Rodrigues, Elias Procópio Duarte Jr., and Luciana Arantes. 2023. $\Diamond P$-vCube: An Eventually Perfect Hierarchical Failure Detector for Asynchronous Distributed Systems. In *12th Latin-American Symposium on Dependable and Secure Computing (LADC 2023), October 16–20, 2023, La Paz, Bolivia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3569902.3569948

1 INTRODUCTION

Although we have grown accustomed to planetary-scale systems and applications, several popular distributed systems abstractions do not scale well [7, 9]. That is certainly the case for most approaches currently used to implement failure detectors, the abstraction proposed by Chandra and Toueg [1] to investigate the solution of consensus in asynchronous systems with crash faults. Most implementations of failure detectors employ an all-to-all approach: every process has to (somehow) communicate with every other process to be able to determine their states. Processes are classified as either correct or suspected of having failed. As the system is asynchronous and there are no timing guarantees, false suspicions can occur. The number of messages employed whenever the underlying network does not support multicast at the hardware level is quadratic. Gossip is an alternative that has been extensively explored to solve this problem [2, 12], but as it is a probabilistic approach, the lack of deterministic guarantees may prevent its applications in several contexts. In this work, we present a scalable, hierarchical strategy to implement failure detectors for asynchronous distributed systems based on the vCube virtual topology, described next.

The virtual Cube – vCube – was originally proposed in the context of hierarchical system-level adaptive diagnosis [5]. The underlying system is assumed to be fully-connected, in the sense that any two pairs of processes can (if needed) communicate directly between themselves, without having to pass through intermediaries. When all processes are correct and the total number of processes (n) is a power of 2, then vCube is a hypercube. However, as processes fail or if n is not a power of 2, the topology self-reconfigures itself, maintaining several logarithmic properties [4] being scalable by definition. vCube has been employed to develop multiple different applications, such as atomic broadcast [18], distributed integrity checking [22], , mutual exclusion [16], and publish/subscribe [3], among others.

Distributed diagnosis implicitly assumes a synchronous model, and that is how the vCube was originally specified. Although there have been abstractions built on top of vCube that assume an asynchronous system (e.g. [11]), they deal with time uncertainties at a higher level. In [6] the authors define the vCube as a failure detector, and investigate under which conditions it guarantees the two classic failure detector properties: completeness and accuracy. Failure detectors classify a process in one of two states: *correct* or *suspected*, assuming a crash model. Informally, completeness reflects the ability of the detector to identify processes that actually failed. Accuracy, on the other hand, is the ability of the detector not to suspect correct processes (false suspicions).

This work presents $\Diamond P$-vCube: an eventually perfect failure detector for asynchronous distributed systems. We show that $\Diamond P$-vCube guarantees not only completeness but also eventual accuracy. In order to deal with false suspicions, which may cause the view of correct processes on other processes to be in a temporarily inconsistent state, a process that detects that it has been suspected leaves the system forever. Furthermore, a process that suspects all others also leaves the system. This approach is similar to the fail-aware failure detectors proposed by [8], where a correct process, once suspected by others, suspects itself. We prove that $\Diamond P$-vCube guarantees eventual accuracy as long as no correct process permanently and simultaneously suspects and is suspected by all its testers, i.e. all correct processes are strongly connected among themselves. The failure detector was implemented with simulation and compared with both the classic solutions based on the popular all-to-all communications, and also with a failure detector based on a ring virtual topology. Results confirm the scalability of vCube, which presents a compromise between how fast actual failures are detected and the cost in terms of the number of messages employed.

The rest of the paper is organized as follows. Section 2 introduces failure detectors and their main properties. Section 3 presents related work. The proposed $\Diamond P$-vCube failure detector is presented in section **??**. Section 5 describes the implementation and simulation results. Section 6 concludes the paper.

2 BACKGROUND: FAILURE DETECTORS

Unreliable failure detectors [1] are used to monitor distributed systems. They detect whether a process or node in the system has failed or is unavailable. The ultimate purpose of failure detectors is to allow distributed systems to be reliable. By detecting failed nodes, the system can take appropriate actions, such as redistributing tasks to other nodes or replicating data to ensure data availability.

Failure detectors can be described as a group of n-detection failure modules, where each of these modules is connected to a different process in the system. These modules cooperate in order to satisfy the failure detector demanded properties [14].

Considering the following completeness properties, as defined in [1]:

Strong completeness Eventually, every process that crashes is permanently suspected by every correct process;

Weak completeness Eventually, every process that crashes is permanently suspected by some correct process.

When considered by itself, completeness is trivial to guarantee by forcing each process to permanently suspect all other system processes, which is not desirable. It is thus necessary to consider another property – accuracy – also defined by [1]:

Eventual Strong accuracy Eventually, no correct process is ever suspected by any other correct process;

Eventual Weak accuracy Eventually, some correct process is never suspected by any correct process.

By combining these properties in pairs, four classes of fault detectors can be defined, as shown in Table 1.

Table 1: Four classes of failure detectors considering eventual accuracy.

Completeness	**Accuracy**	
	Eventual Strong	**Eventual Weak**
Strong	Eventually Perfect $\Diamond P$	Eventually Strong $\Diamond S$
Weak	Eventually Quasi-Perfect $\Diamond Q$	Eventually Weak $\Diamond W$

The $\Diamond P$ and $\Diamond S$ detectors guarantee that all faulty processes will be detected by all correct processes (strong completeness) and that, in a finite time, false suspicions will cease to happen for all or some correct process, respectively.

There are different alternatives for implementing a failure detector. In this work, we employ the *push-based* strategy, according to which processes test each other to determine their states. The most common strategy is the so-called *pull-based* or heartbeat-based strategy. A process monitored by a heartbeat-based failure detector [10] sends periodic messages (called heartbeats) with the purpose to inform that it is correct. If the failure detector does not receive a heartbeat (or a number of heartbeats) within the expected time interval, it considers the monitored process has failed - the process gets to be suspected. Yet another approach is that of suspicion-based failure detectors, which rely on the idea of detecting the failure of a process only based on the suspicion level of other nodes in the system. Each node keeps track of the status of other nodes and raises the suspicion level if it detects any anomalies.

In an asynchronous system model, there is no assumption of any upper bound on message delivery delays, and any processes can crash. This makes failure detection more challenging compared to synchronous system models, where message delivery delays are bounded and failure detection can be done using timeouts. However, it is worth noting that failure detectors are not foolproof, and there is always a chance of false positives or false negatives. It is important to carefully design and tune the failure detector to balance the risks of false positives and false negatives.

In the asynchronous model, failure detectors typically rely on additional assumptions or techniques to overcome the lack of timing guarantees. For instance, the heartbeats of heartbeat-based failure detectors can be delayed, leading to false suspicions. A smart time-out can be employed to predict such a delay and avoid false suspicion.

In another approach, the Impact failure detector [17] is designed to provide accurate failure detection while minimizing the number of false positives and false negatives. The Impact failure detector uses a probabilistic approach that takes into account the impact of each node's failure on the system's availability. This information is used to estimate the probability of each node's failure and update the suspicion level accordingly.

3 RELATED WORK

In [1] the authors presented a detection algorithm of *timeouts* implementing a $\Diamond P$ detector. The algorithm uses *heartbeat* messages periodically exchanged between all processes, which requires a quadratic number of messages and a partially synchronous system.

To reduce the number of detection messages, [13] proposed a ring-based solution for partially synchronous systems. Each process monitors only the next correct process in the ring, i.e., its direct successor. Therefore, the number of messages sent periodically is linear. It is a $\Diamond P$ class detector, i.e., faulty processes are permanently suspected, but false suspicions may occur during a period of instability.

Later, [14] proposed a new version of the algorithm implementing a $\Diamond S$. The ring organization was retained, but instead of each process testing its successor, the system tries to determine a correct process that is common to all others. In this case, only the common correct process sends "I_AM_ ALIVE" messages at regular intervals. When the trusted process stops sending messages, the next process in the ring becomes the trusted process. Monitoring is done by *timeouts* and the time interval varies for each monitored process and is increased when a false suspicion is detected. The version was compared with the solution of [1] and [13] and proved to be more efficient in terms of the number and the size of messages.

SWIM [2] is a fault detection solution that aims to increase the scalability of detection protocols. Unlike traditional protocols based on *heartbeats*, SWIM separates the functionality of error detection and dissemination of membership updates from the *membership* protocol. The processes are monitored by a random point-to-point probe. The expected time for initial detection of a process failure and the expected message load per member does not vary with group size. Information about membership changes, such as process joins, terminations, and failures, is piggybacked in *ping* and *acks* messages.

In [21], two atomic broadcast algorithms were compared, based on unreliable failure detectors and group management protocols. The original $\Diamond S$ failure detector algorithm proposed by Chandra and Toueg was employed [1].

Falcon [15] is a failure detector that uses a network of spy modules, or spies, which employ internal information to determine whether the system modules are active or not. When a module appears to have failed, the spies stop it for good, allowing the detector to report that the process has failed without error. The system has been implemented and evaluated in monitoring applications, operating systems resources, virtual machines and network switches.

The solution proposed by Fetzer and Cristian [8] is particularly related to the present work. The authors introduced fail-aware failure detectors, also defining strong and weak fail-awareness properties. Strong fail-awareness requires a process to suspect itself as soon as another process suspects it. Weak fail-awareness requires a failure detector to suspect its associated process whenever it is suspected by a majority of processes.

4 THE $\Diamond P$-VCUBE FAILURE DETECTOR

A distributed system is defined as a finite set P with $n > 1$ processes $\{p_0, .., p_{n-1}\}$ that communicate by exchanging messages. The operations to send and receive messages are both atomic. The links connecting processes are reliable. The system is represented by a complete graph, i.e., every pair of processes can communicate directly without the need for intermediaries. Furthermore, processes form a vCube overlay, as described below.

The system is asynchronous, namely, there are no limits on message transmission delays and on the relative processing speeds of the processes. The permanent crash fault model is assumed. A faulty process no longer sends out any messages. A process that never crashes and replies correctly to the detector is considered to be *correct* or *fault-free*. Otherwise, the process is *faulty* or *suspected*.

$\Diamond P$-vCube is a push-based failure detector. Thus, instead of employing heartbeats, a process executes a test on another process to determine its state. A test consists of the exchange of messages between a tester and a tested node. If a test succeeds, both the tester and tested processes identify each other as correct. The virtual edges of a vCube correspond to the tests that the correct processes perform on each other. The set V of processes of the system P test each other forming a virtual topology that corresponds to directed graph $A = (V, T)$, where T is the set of tests executed.

The dimension of a vCube is defined as $\lceil \log(n) \rceil$, all logarithms in this work are base 2. Processes of a vCube with dimension $d > 0$ have identifiers consisting of d bits. In a faultless scenario, two processes are virtually connected if their binary addresses differ by a single bit. In the example in Figure 1, process 1 (001) is connected to processes 0 (000), 3 (011), and 5 (101).

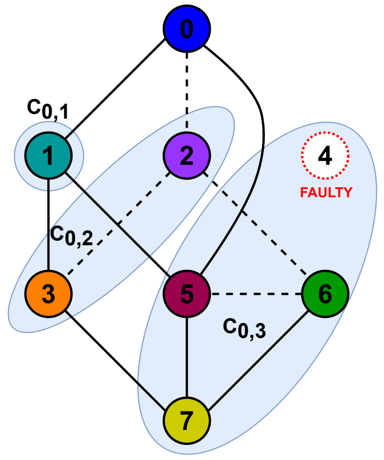


Figure 1: Clusters of a vCube with $2^3 = 8$ processes; p_4 is faulty.

vCube organizes processes into progressively larger clusters of 2^{s-1} processes, $s = 1, .., \log_2 n$. Function $c_{i,s}$ (Equation 1) returns the sorted list of processes in each cluster, where $\oplus$ is the exclusive *bitwise* operator (*XOR*).

$$c_{i,s} = \{i \oplus 2^{s-1},\ c_{i \oplus 2^{s-1}, 1}, \dots,\ c_{i \oplus 2^{s-1}, s-1}\} \quad (1)$$

Table 2 shows the outcomes of the $c_{i,s}$ function for 8 processes. To determine the edges of the virtual topology, for each node i there is an edge (j, i) such that j is the first fault-free node in

$c_{i,s}$, $s = 1 ... \log_2 n$. After a process detects that another process has failed, the set of edges (tests) is recomputed. For example, in Figure 1, process p_0 originally tests process p_4 in *cluster* 3, but after p_4 fails, process p_0 tests process p_5, which is next in the considered correct $c_{0,3}$ list.

Table 2: Result of function $c_{i,s}$ for 8 processes.

s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2,3	3,2	0,1	1,0	6,7	7,6	4,5	5,4
3	4,5,6,7	5,4,7,6	6,7,4,5	7,6,5,4	0,1,2,3	1,0,3,2	2,3,0,1	3,2,1,0

Algorithm 1 presents the pseudocode of the proposed $\diamond P$-vCube failure detector for asynchronous systems. Processes running $\diamond P$-vCube execute tests periodically, on a testing interval defined using the local clock. Process i keeps track of the state of all other processes in the $STATE_i[]$ array. Note para process i also obtains from a tested correct process j the information that j maintains on the state of i. Thus process i can identify whether it has been the subject of a false suspicion, i.e. process i was suspected erroneously by process j. In this case, process i, stops executing and leaves the system (line 8). The result is a $\diamond P$ failure detector, as shown below.

A testing round has occurred when all correct processes have executed all their assigned tests. The failure detection latency is defined as the number of rounds it takes for all correct processes to detect a state change. The latency for the vCube to detect a process failure is a logarithmic function of the total number: $\log_2^2 N$ testing rounds in the worst case, but much lower in average [4].

Next, we prove that $\diamond P$-vCube is an eventually perfect failure detector. Theorem 4.1 proves that $\diamond P$-vCube guarantees eventual strong accuracy.

Theorem 4.1. *Even if there are false suspicions, $\diamond P$-vCube always guarantees the eventual strong completeness property.*

Proof. Consider that process j has failed. Now any process i that tests j determines correctly suspects j, as it has permanently crashed and does not send any reply to a test. If some process k does not test j, then if there is a path on A consisting only of correct processes from k to any other process that tested i and information about j's failure will reach k in at most $\log_2^2 N$ testing rounds. If there is no path in A from k (that does not j) to j, then $\diamond P$-vCube guarantees that k eventually tests and suspects j.

□

Theorem 4.2. *$\diamond P$-vCube guarantees eventual strong accuracy.*

Proof. In case no correct process permanently and simultaneously suspects and is suspected by any of its testers, all correct processes are strongly connected among themselves, all correct processes will determine their correct state.

Now, consider that at least one of the testers of a correct process j, say process i, suspects j, while at least another tester, say process k does not suspect j. In this case, j does not suspect i either, the test which consists of the exchange of messages between processes has succeeded. However, as i and k are strongly connected between themselves, k will obtain the information that i suspects j and will update $STATE_k[j]$. The next time k succeeds in testing j, they exchange information, and j will learn it has been falsely suspected and will halt execution.

In case j permanently and simultaneously suspects and is suspected by all its testers, which eventually consists of all system nodes, then it also halts, guaranteeing the eventual strong accuracy. □

5 SIMULATION & RESULTS

Neko [20] is a Java *framework*[1] that was developed with the goal of enabling the simulation of distributed algorithms and evaluating their performance. Its architecture is divided into two main layers: Application and Network. An application is built in terms of micro protocols. Micro protocols are registered in processes (*containers*), which are instances of the NekoProcess class. At the application level, processes communicate by passing messages. Messages are sent and received using the send and deliver methods, respectively. Micro protocols can communicate in two ways: over the network if they are registered in different processes, or by calling the corresponding method by direct reference if they belong to the same process. If the message is sent over the network, once it is received by the target process, it is delivered directly to the micro protocol specified in the message header. In a simulation, all processes reside on a single computer. In a distributed execution, each process may be on a different computer.

The second component of Neko's architecture is the network, which can be simulated or real. Simulated networks include two main implementations:

- BasicNetwork Uses a parameter `lambda` to create fixed transmission delays;
- RandomNetwork Generates random transmission delays (random from Java) based on a parameter `lambda` that varies between 0 and 1. The `seed` parameter can be used to reproduce a particular experiment.

A real network uses *sockets* TCP communication and must be instantiated by running modules on each host. In this way, processing and communication time is obtained from the real environment.

5.1 Support for simulation of failures and false suspicions

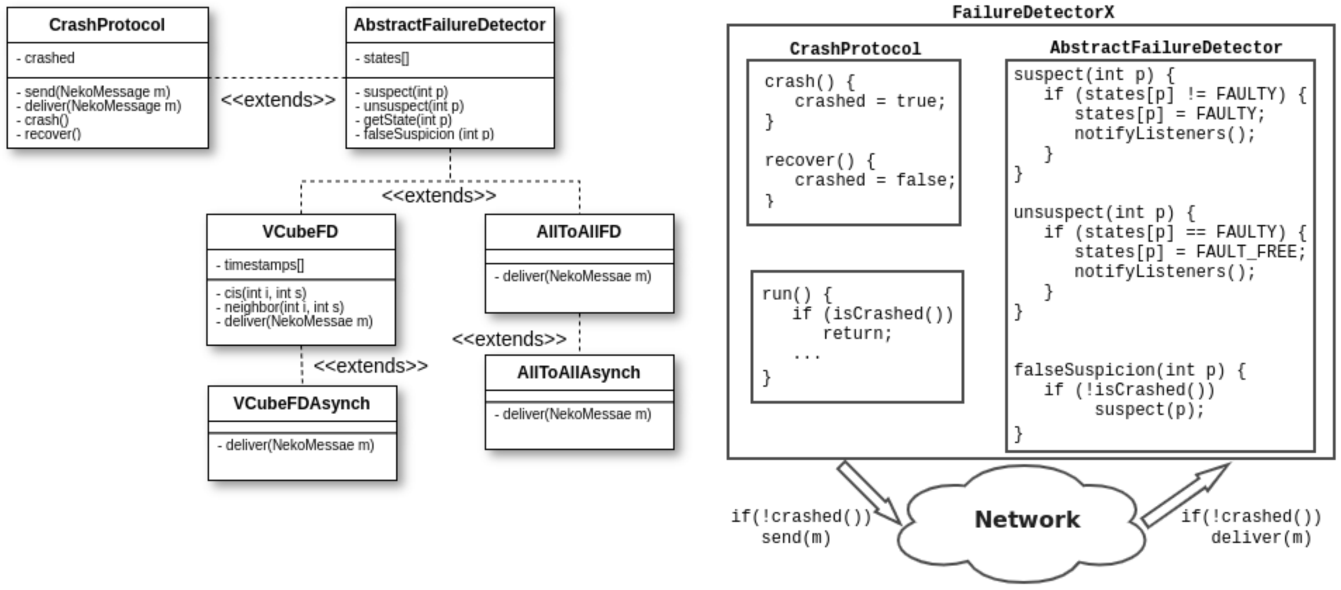
Simulation of failures in Neko is performed using the adaptations proposed in Figure 2. A collapse mechanism starts and stops failure intervals according to the configuration file used for other Neko configurations. The application sends messages to the failure simulation support class, which checks if the process collapses. If so, the message is discarded. The same applies to messages received from the network. The application can query the status of the process and in case of an error stop its execution by a `crashed` flag.

In this work, false suspicion simulation mechanisms have been included. The `AbstractFailureDetector` class now provides a `falseSuspicion(`*intp*`)` method that receives as a parameter the identifier of the p process that is suspected by the i process. The method checks whether the process is in a valid execution state, i.e.

[1]source code available at https://github.com/arluiz/neko

Algorithm 1 The $\Diamond P$-vCube failure detector executed by process i

```
1:  $STATE_i[j] \leftarrow correct, \forall j = 0, .., n-1$
2:  repeat
3:      for $s \leftarrow 1$ to $\log_2(n)$ do
4:          for all $j \in c_{i,s} \mid i$ is the first correct process $\in c_{j,s}$ do
5:              TEST(j)
6:              if $j$ is tested $correct$ then $i$ obtains $STATE_j[]$ and $j$ obtains $STATE_i[]$
7:                  if $STATE_j[i] = suspect$ then        ▷ $i$ was suspected by $j$ (false suspicion)
8:                      Halt execution and leave the system
9:                  else
10:                     Update $STATE_i[]$ with the received $STATE_j[]$
11:             else
12:                 if $STATE_i[j] = correct$ then $STATE_i[j] \leftarrow suspect$
13:                 if $\forall j = 0, .., n-1, j \neq i$: $STATE_i[j] = suspect$ then
14:                     Halt execution and leave the system
15:     Sleep until the next testing interval
16: until forever
```



```
crash() {
   crashed = true;
}

recover() {
   crashed = false;
}
```

```
run() {
   if (isCrashed())
      return;
   ...
}
```

```
suspect(int p) {
   if (states[p] != FAULTY) {
      states[p] = FAULTY;
      notifyListeners();
   }
}

unsuspect(int p) {
   if (states[p] == FAULTY) {
      states[p] = FAULT_FREE;
      notifyListeners();
   }
}


falseSuspicion(int p) {
   if (!isCrashed())
      suspect(p);
}
```

```
if(!crashed())
  send(m)
```

```
if(!crashed())
  deliver(m)
```

Figure 2: Neko crashes and false suspicions simulation modules.

it is not faulty in the simulator context and calls the `suspect(`p`)` method, which was already part of Neko. This method is called via a parameter:

```
process.i.false_suspect.process.p =< time >
```

which must be included in the simulation's general configuration file. The source code and a configuration example can be found at https://github.com/arluiz/vcubefd-asynch.

5.2 vCube Vs. vRing Vs. All-to-All

In order to illustrate the advantages of employing vCube in comparison with other alternatives, Figure 3 different test runs of different failure detectors all executed with the Neko [20] simulator. Figure 3(c) shows a vCube test run. Also shown is the test run for the classic all-to-all implementation of failure detectors (each process monitors all others) (Figure 3(a)). Furthermore, a failure detector based on a virtual ring (vRing) defined in [6] is also shown (Figure 3(b)). In the all-to-all strategy, each process tests all other $n-1$ processes, resulting in a total of $n(n-1)$ tests, which is quadratic. In the vRing approach, each process i tests process $i+1$ which is the next in the ring, and so on until a correct process is found. vRing employs at most n messages ($2*n$ in a request-reply strategy). In the case of vCube, each process is tested by at most $\log_2 n$ neighbors, for a total of $n * log_2 n$ tests in the worst case. The first sequence of messages represents ARE_YOU_ALIVE requests and the second, I_AM_ALIVE responses. In this case, the number of messages is twice the number of tests.

A testing round has occurred when all correct processes have executed all their assigned tests. The failure detection latency is defined as the number of rounds it takes for all correct processes to detect a state change. The latency for the vCube to detect a process failure is a logarithmic function of the total number: $\log_2^2 N$ testing rounds in the worst case, but much lower in average [4]. Meanwhile, the all-to-all strategy presents a latency of a single testing round, and vRing has a latency of n testing rounds. Thus the vCube represents a compromise between how fast actual failures are detected and the cost in terms of the number of messages employed.

The performance of distributed algorithms is commonly measured based on two metrics: time complexity and [19] message complexity. The time complexity measures the latency of the algorithm, that is, the execution time. Message complexity consists of counting the total number of messages generated by the algorithm.

In this work, latency was evaluated in two aspects: 1)*crash* failure: diagnostic latency of the *crash* failure, that is, after the failure, how much time is required for all other processes to be notified; 2) false suspicions: a) time for the process to identify that it was suspected incorrectly; b) false suspicion detection latency: it is the time for all processes to identify a process as faulty based on the false suspicion reported by the first process.

The algorithms were evaluated in three scenarios: a) without failures or false suspicions; b) with *crash* failures; and c) without failures, but with false suspicions. For each scenario, systems with $n = 2^d$ processes were used, for $d = 2, 3, .., 8$, i.e., $n = 4, 8, 16, .., 256$. Failure parameters are described in the following scenarios, where applicable.

The two Neko network models used were the BasicNetwork and the RandomNetwork. For each message sent, a sending time of 0.1 time intervals is considered, a message transmission time over the network and a reception time of 0.1. In simulations with the BasicNetwork network, the time interval between sending and receiving a message is 1.0, but for each message sent in sequence, the sending time is shifted by 0.1 (As illustrated in Figure 3). As the receiving is controlled by the simulator, the transmission time on the network was configured by the parameter `BasicNetwork.lambda=0.9`. On the RandomNetwork, the transmission time varies randomly according to a `RandomNetwork.lambda` parameter.

vCube was compared to a classic all-to-all (ALL) solution, in which each correct process sends an individual test to all other processes considered correct, and with a Virtual Ring (vRing) solution, where the processes are all fully connected but each correct process sequentially tests until it finds the next correct process. For a fair comparison, in case of a false suspicion, the suspected process also leaves the system in all-to-all and vRing.

All experiments were performed in $\log_2^2(n)$ rounds, which is the vCube's maximum diagnostic latency in the worst case, in order to guarantee that all processes will be tested by all others. The testing interval for both algorithms was set to 30.0. At each testing interval, vCube only tests the neighbors.

5.3 No failures or false suspicions

In this first test scenario, the network models BasicNetwork and RandomNetwork were used to simulate scenarios in which there were no failures of any of the processes.

In Figure 4, it is possible to observe the execution time of all rounds and the number of messages of the vCube algorithm all-for-all (ALL) and the virtual ring (vRing), after being executed in a BasicNetwork model. The execution time varies for the three algorithms, but increases faster for ALL as the number of processes increases. vRing has the same execution time as VCube, since the time to go around the ring is equivalent to the propagation time between the source and the farthest leaf of the hypercube. The number of messages for ALL is also noticeably higher, being $n \log_2 n$ tests per round for vCube, n^2 for ALL, and n for vRing, due to the sequential testing. Each test uses a request (REQUEST) and response (REPLY) strategy.

The second scenario tested was a network model RandomNetwork with transmission delay parameter `lambda=0.1`, which generates random delays, but without large variations, thus avoiding false suspicions. The result is exactly the same as the previous scenario, except for small variations in transmission delays. For the sake of space, graphics have been omitted.

Table 3 shows the values for the number of messages in the two scenarios without failures. Regardless of the type of network, the total number of messages is the same. As expected, ALL exchanges a much larger number of messages compared to vCube and vRing, especially when the number of processes grows. vRing sends fewer messages than both others.

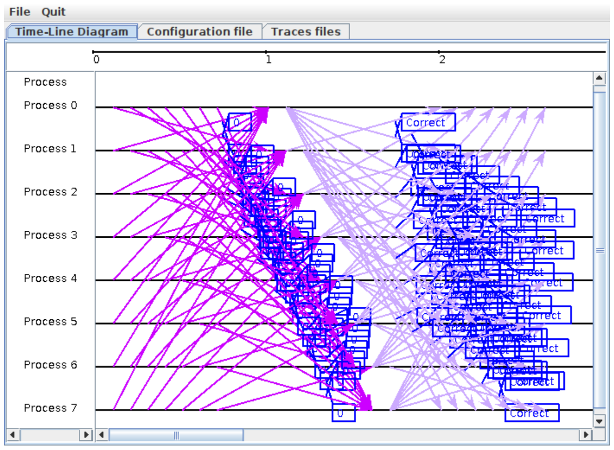
Table 3: Number of messages to ALL, vCube and vRing in fault-free scenarios.

Processes	ALL	vCube	vRing
4	96	64	32
8	1.008	432	144
16	7.680	2.048	512
32	49.600	8.000	1.600
64	290.304	27.648	4.608
128	1.593.088	87.808	12.544
256	8.355.840	262.144	32.768

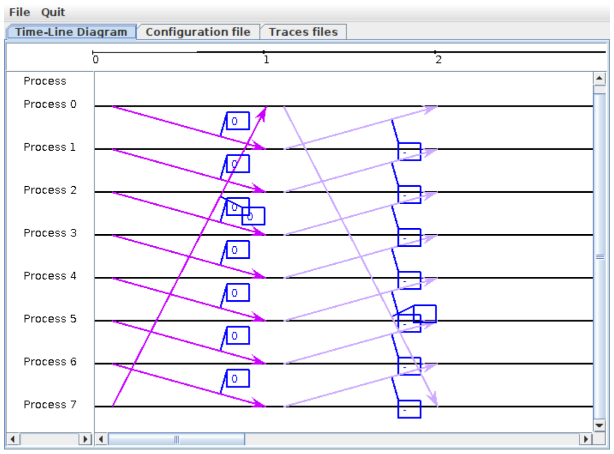
5.4 Crash failures

To simulate a *crash* failure, the Neko mechanism was used in which it was possible to define a failure signal of process 0 at time 0 (zero) of the simulation. The vCube, ALL and vRing algorithms were executed on the BasicNetwork network model. In the scenario with a single failure, the execution time is very close to the scenario without failures, since each execution includes more than one round of tests and the detection latency is diluted in the total time. The same happens with the number of messages, although it is slightly lower as the processes detect the failure and stop testing the failed process.

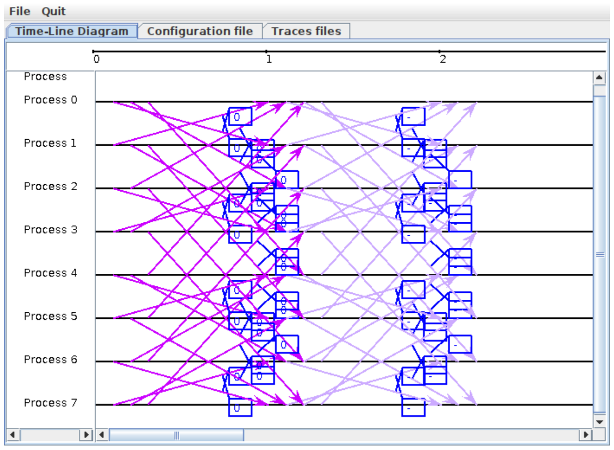
Figure 5 shows the latency of a failure *crash* started at time 0, that is, the time interval between the failure of a process and



(a) All-to-All ($O(n^2)$ messages)



(b) vRing ($O(n)$ messages)



(c) vCube ($O(n)$ messages)

Figure 3: All-for-all, vRing and vCube testing messages for 8 processes (generated by Neko's LogView tool [20]).

the identification of the failure in all the correct processes. As process 0 is the first to be tested by all processes in ALL, the time is constant and is linked to the *timeout* of the test. This value could be proportionately larger if the failed process was the last one to be tested due to the transmission delay of sequential messages. Even so, the detection would happen in the same round of tests.

The latency of vCube is proportional to the number of test runs. In the first round, only neighbors connected virtually to the failed process identify the event. In the second round, neighbors two hops away identify the failure, and so on. The diagnostic latency of the vRing algorithm is directly related to the number of processes in the system. Since only one process learns of the failure in each round of testing, it takes up to $n-1$ rounds for all correct processes to identify the faulty process.

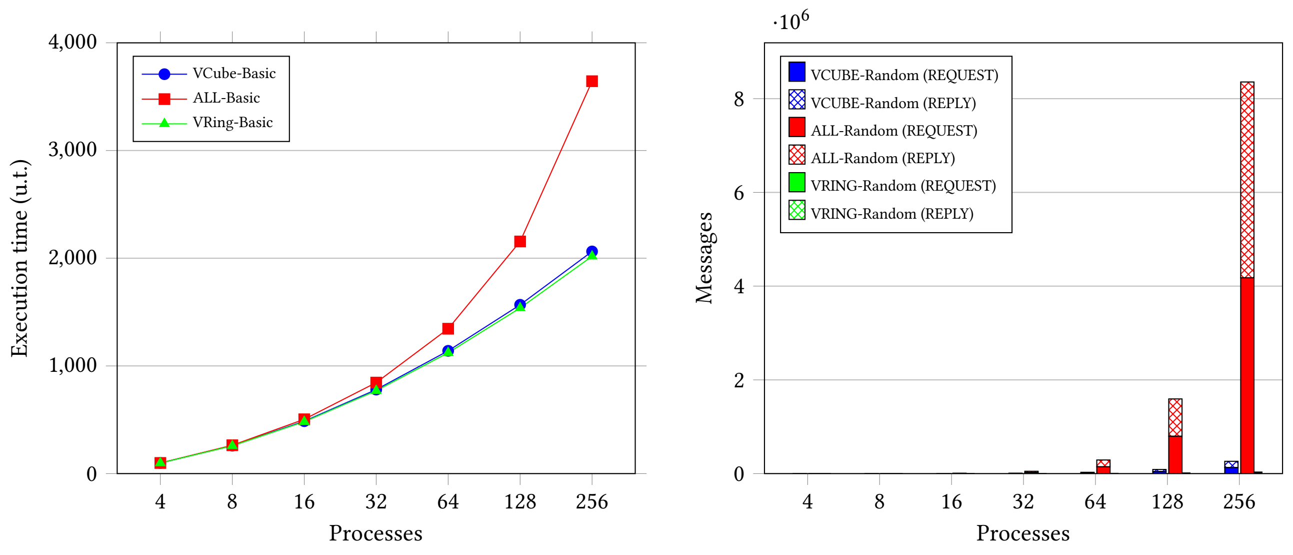
Therefore, diagnostic latency is always higher in vCube compared to all-to-all and always lower than vRing. This result is comparable when false suspicions occur, since information propagation follows the same strategy.

5.5 False suspicions

The scenario with the introduction of false suspicions using the new resource implemented in Neko behaves similarly to the *crash* scenario. Figure 6 illustrates an example of *log* generated for a system with 4 processes. Process 0 was configured to suspect process 1 at time 0. In the first line, process 0 already signals that it suspected process 1. Therefore, it starts directly testing process 2 (line 3). Process 1 tests process 0 and receives the information that it is suspected of it (lines 4 and 10, respectively). Upon receiving this information, process 1 leaves the system (line 11) and, on line 12, a *crash* signal is created by it to stop its execution.

A second way to test false suspicions is to use the RANDOMNETWORK network. In this case, `lambda = 0.6` was used, which generates large variations in the network transmission time, causing premature *timeouts* and, consequently, false suspicions.

Figure 7 presents the total execution time and the number of messages for the three tested algorithms. The total execution time of the $\log_2^2 n$ rounds in each scenario is less than in the previous



(a) Execution Time (u.t.)   (b) Messages

Figure 4: Fault-free execution on BasicNetwork.

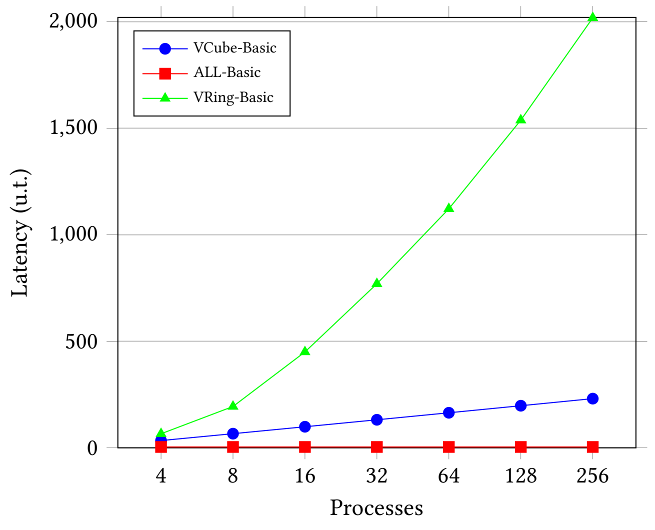


Figure 5: Diagnosis latency for one fault in the time 0.0 on the BasicNetwork.

```
0,000 p0 messages falsely suspect p1
0,000 p0 messages suspect 1
0,100 p0 messages e s p0 p2 ARE_YOU_ALIVE 0
0,100 p1 messages e s p1 p0 ARE_YOU_ALIVE 0
..
1,000 p0 messages e r p1 p0 ARE_YOU_ALIVE 0
..
1,100 p0 messages e s p0 p1 I_AM_ALIVE true
..
2,000 p1 messages e r p0 p1 I_AM_ALIVE true
2,000 p1 messages p0 suspect me: die!
2,000 p1 messages crash started at crash-all2all-fd
```

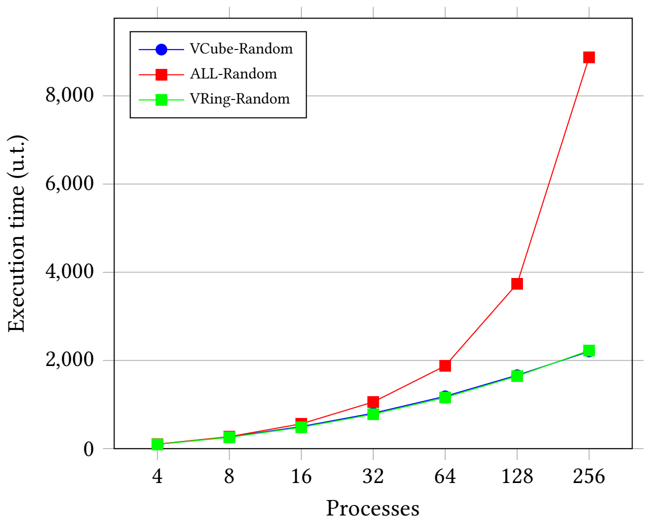
Figure 6: Clipping of *log* generated for a false suspicion of process 1 by process 0 on network BasicNetwork.

Table 4: Number of suspected processes on the RandomNetwork network with `lambda=0.6`.

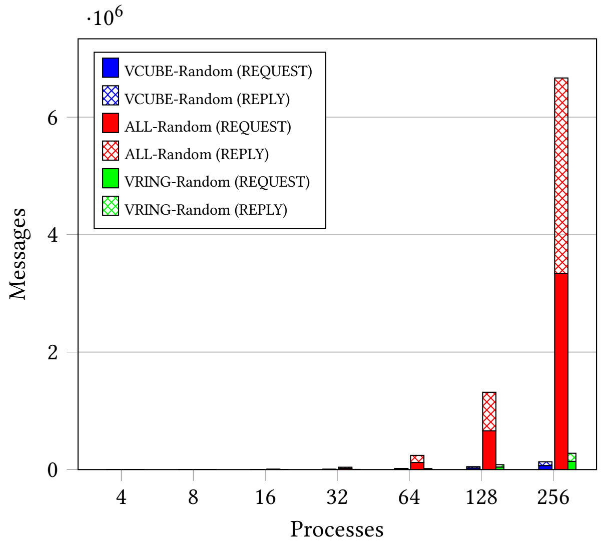
Processes	ALL	vCube	vRing
4	1	1	0
8	1	2	1
16	2	5	3
32	5	13	6
64	7	33	22
128	14	82	40
256	35	187	154

executions, especially for the vCube. This is due to the number of processes that were suspected and left the system, as can be seen in Table 4.

For the same reason, in Figure 7(b), it is possible to identify that there is a significant reduction in the amount of messages exchanged by processes. This is because this scenario produces many suspected processes, which leave the network when identifying themselves as suspected. As a result, fewer tests are performed both by those who left the network and by those who detected the failure and stopped testing the suspect process. Although it is the behavior of the network, it is not a fair comparison because vCube and vRing had many more suspected processes than ALL.



(a) Execution time (u.t.)



(b) Messages

Figure 7: Execution with over one failure in RandomNetwork **with** `lambda=0.6`**.**

6 CONCLUSIONS AND FUTURE WORK

This work presents $\Diamond P$-vCube, an eventually perfect hierarchical push-based failure detector for failure detection in asynchronous systems. In those systems false suspicions can occur, due to unbound message delays and process execution times. In the proposed algorithm, a correct process halts itself after learning that it has been identified as suspicious by any other process or suspects all other processes. This strategy guarantees the eventual accuracy of the proposed detector. We have also shown that $\Diamond P$-vCube guarantees the eventual strong completeness.

$\Diamond P$-vCube was compared using simulation to the classic all-to-all implementation of failure detectors, as well as with a solution based on a virtual ring (vRing). To simulate false suspicions, a new mechanism was built into the Java Neko *framework*. That mechanism can be used to configure when one process should suspect another, regardless of its current state. False suspicions can also be generated by varying the transmission times of the simulated network using features available in the simulator. Results show that $\Diamond P$-vCube has a failure detection latency higher than all-to-all but lower than that of vRing. In terms of execution time and number of messages, it has been shown that $\Diamond P$-vCube uses a considerably lower number of messages than all-to-all, which also reduces the execution time of the test runs, with all-to-all having the highest execution time. vRing requires a considerably lower number of messages than $\Diamond P$-vCube, but the execution times of the test runs is very similar to $\Diamond P$-vCube.

To conclude, $\Diamond P$-vCube represents the middle ground between the other two solutions, clearly showing a compromise between how fast actual failures are detected and the cost in terms of the number of messages employed. Future work includes the investigation of robust distributed applications on top of $\Diamond P$-vCube, such as consensus and reliable/ordered broadcasting.

ACKNOWLEDGMENTS

This work was partially supported by the Brazilian Research Council (CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico) grant 308959/2020-5; FAPESP/MCTIC/CGI grant 2021/06923-0; and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil) - Finance Code 001.

REFERENCES

[1] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647

[2] A. Das, I. Gupta, and A. Motivala. 2002. SWIM: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*. 303–312. https://doi.org/10.1109/DSN.2002.1028914

[3] João Paulo de Araujo, Luciana Arantes, Elias P. Duarte, Luiz A. Rodrigues, and Pierre Sens. 2017. A Publish/Subscribe System Using Causal Broadcast over Dynamically Built Spanning Trees. In *29th SBAC-PAD*. 161–168. https://doi.org/10.1109/SBAC-PAD.2017.28

[4] E. P. Duarte, L. C. E. Bona, and V. K. Ruoso. 2014. VCube: A Provably Scalable Distributed Diagnosis Algorithm. In *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 17–22. https://doi.org/10.1109/ScalA.2014.14

[5] Elias P Duarte and Takashi Nanya. 1998. A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Transactions on computers* 47, 1 (1998), 34–45.

[6] Elias P Duarte Jr, Luiz A Rodrigues, Edson T Camargo, and Rogerio Turchetti. 2022. A Distributed System-level Diagnosis Model for the Implementation of Unreliable Failure Detectors. *arXiv preprint arXiv:2210.02847* (2022).

[7] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[8] C. Fetzer and M. Cristian. 1996. Fail-aware failure detectors. In *Proceedings 15th Symposium on Reliable Distributed Systems*. 200–209. https://doi.org/10.1109/RELDIS.1996.559722

[9] Ian Gorton. 2022. *Foundations of Scalable Systems*. " O'Reilly Media, Inc.".

[10] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. 2004. The /spl phi/ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004*. 66–78. https://doi.org/10.1109/RELDIS.2004.1353004

[11] Denis Jeanneau, Luiz A. Rodrigues, Luciana Arantes, and Elias Procópio Duarte Jr. 2017. An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection. *J. Braz. Comput. Soc.* 23, 1 (2017), 15:1–15:14. https://doi.org/10.1186/s13173-017-0064-9

[12] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. 2005. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)* 23, 3 (2005), 219–252.

[13] Mikel Larrea, Sergio Arevalo, and Antonio Fernndez. 1999. Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. In *Distributed Computing*, Prasad Jayanti (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 34–49.

[14] M. Larrea, A. Fernandez, and S. Arevalo. 2000. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*. 52–59. https://doi.org/10.1109/RELDI.2000.885392

[15] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. 2011. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. 279–294.

[16] Luiz A. Rodrigues, Elias P. Duarte, and Luciana Arantes. 2018. A distributed k-mutual exclusion algorithm based on autonomic spanning trees. *J. Parallel and Distrib. Comput.* 115 (2018), 41–55. https://doi.org/10.1016/j.jpdc.2018.01.008

[17] Anubis Graciela de Moraes Rossetto, Cláudio F R Geyer, Luciana Arantes, and Pierre Sens. 2018. Impact FD: An Unreliable Failure Detector Based on Process Relevance and Confidence in the System. *Comput. J.* 61, 10 (04 2018), 1557–1576. https://doi.org/10.1093/comjnl/bxy041 arXiv:https://academic.oup.com/comjnl/article-pdf/61/10/1557/25949207/bxy041.pdf

[18] Lucas V Ruchel, Luiz Antonio Rodrigues, Rogério C Turchetti, Luciana Arantes, Elias P Duarte Jr, and Edson T Camargo. 2022. A Leaderless Hierarchical Atomic Broadcast Algorithm. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing*. 61–66.

[19] P. Urban, X. Defago, and A. Schiper. 2000. Contention-aware metrics for distributed algorithms: comparison of atomic broadcast algorithms. In *9th Int'l Conf.on Computer Communications and Networks (ICCCN)*. 582–589. https://doi.org/10.1109/ICCCN.2000.885548

[20] P. Urban, X. Defago, and A. Schiper. 2001. Neko: a single environment to simulate and prototype distributed algorithms. In *Proceedings 15th International Conference on Information Networking*. 503–511. https://doi.org/10.1109/ICOIN.2001.905471

[21] P. Urban, I. Shnayderman, and A. Schiper. 2003. Comparison of failure detectors and group membership: performance study of two atomic broadcast algorithms. In *Int'l Conf. on Dependable Systems and Networks*. 645–654. https://doi.org/10.1109/DSN.2003.1209974

[22] R.P. Ziwich, E.P. Duarte, and L.C.P. Albini. 2005. Distributed integrity checking for systems with replicated data. In *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, Vol. 1. 363–369 Vol. 1. https://doi.org/10.1109/ICPADS.2005.130