



UNIVERSITÀ DEGLI STUDI  
DI TRENTO

## Distributed Systems Course Project: Consensus with Failure Detector

Mattia Avancini  
138793  
avancini.mattia@gmail.com

Giampaolo Farina  
142779  
giampaolo.farina@gmail.com

### Abstract

*The content of this work is about the implementation of the consensus protocol making use of the PeerSim Simulator [5]. Since solving consensus in an unreliable asynchronous distributed system is impossible, even if there is at most one failure and the links are reliable, we need to introduce failure detectors in order to solve it. This report is created as course project relative to the Distributed Systems course held at the University of Trento by prof. Alberto Montresor and his assistant Gianluca Ciccarelli. Our approach at the problem starts introducing consensus, giving and explaining all the characteristics that it presents and then discovering the different kinds of unreliable failure detectors that we can build in order to solve the consensus problem. In the second part of the report we will propose some implementations based on the solution proposed in [1, 6]. These possible solutions are implemented using the PeerSim Simulator framework<sup>1</sup> in order to run some simulations. The last and final step is related to the analysis of the result obtained running the simulations of the protocol we have built.*

---

<sup>1</sup><http://peersim.sourceforge.net/>

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Objectiveness . . . . .	3
1.2	Content . . . . .	3
1.3	Scenario . . . . .	3
<b>2</b>	<b>The consensus problem</b>	<b>4</b>
2.1	Introduction . . . . .	4
2.2	System model . . . . .	4
2.3	Consensus properties . . . . .	4
2.4	Consensus protocol . . . . .	5
2.5	Failure Detectors . . . . .	5
2.5.1	Completeness property . . . . .	6
2.5.2	Accuracy property . . . . .	6
2.5.3	Failure Detectors Classes . . . . .	6
<b>3</b>	<b>Implementation</b>	<b>7</b>
3.1	Structure of the code . . . . .	7
3.2	Messages . . . . .	7
3.3	FDs implementation . . . . .	8
3.3.1	Strong Failure Detector . . . . .	8
3.3.2	Eventually Strong Failure Detector . . . . .	10
3.4	Consensus implementation . . . . .	13
3.4.1	Consensus using strong failure detector . . . . .	13
3.4.2	Consensus using eventually strong failure detector . . . . .	13
<b>4</b>	<b>Simulation</b>	<b>13</b>
4.1	Configuration files . . . . .	14
4.1.1	Network Parameters . . . . .	14
4.1.2	Simulation Engine Parameters . . . . .	14
4.1.3	Protocol Parameters . . . . .	14
4.1.4	Initialization Parameters . . . . .	15
4.1.5	Control Parameters . . . . .	15
4.1.6	consensus_strong.cfg . . . . .	16
4.1.7	consensus_ev_strong.cfg . . . . .	16
4.2	Initializer . . . . .	16
4.3	Observer . . . . .	16
4.4	Running a simulation . . . . .	16
4.5	Running multiple simulations . . . . .	16
<b>5</b>	<b>Analysis and results</b>	<b>17</b>
5.1	Graphs generation . . . . .	18
5.2	Analysis and comments . . . . .	18
5.3	Consensus with strong failure detector . . . . .	19
5.4	Consensus with eventually strong failure detector . . . . .	19
<b>6</b>	<b>Conclusion</b>	<b>19</b>

<b>A Figures</b>	<b>20</b>
A.1 Consensus with Strong FD . . . . .	26
A.2 Consensus with Eventually Strong FD . . . . .	44
<b>References</b>	<b>54</b>

# 1 Introduction

## 1.1 Objectiveness

The aim of this work is to show a possible model for implementing the consensus protocol in an unreliable asynchronous distributed system introducing unreliable failure detectors as suggested in [1]. We made use of the PeerSim Simulator engine [5] in order to develop, test and analyse the behaviour of our proposed solution.

## 1.2 Content

The report is composed by many sections, initially we will examine some theoretical aspects about the consensus problem; then we will describe in details our proposed solution: the protocols that we have implemented and the analysis that we have made. In particular we handle the following topics:

- In the first chapter we will describe the consensus problem from a theoretical point of view. As mentioned before to solve the consensus problem in an asynchronous system we need to introduce distributed oracles called failure detectors. Hence, we will illustrate the characteristics of the various kinds of failure detectors used in our proposed solution.
- In the second section of the report we are going into depth examining the implementation details of the proposed model. Firstly we will introduce a general overview of the system architecture and code structure. Then all the system components we use to build our algorithm are explained in details: such as the various messages and channels used, the failure detectors that we chose to implement and finally the core of the system, the consensus algorithm.
- The third chapter shows the components offered by the PeerSim framework introduced to perform the simulation, like the initiators and the observers. Inside the chapter we will explain how to run a particular simulation.
- The last chapter instead describes and try to explain the results obtained from the simulations and mainly treats the different results in terms of varying the configuration parameters. Inside the chapter we will discover how we perform and conduct the analysis on the system behaviour and performance. For example we will explain what are the kind of graphs generated to help us understand the behaviour of the protocol and how to obtain them.

## 1.3 Scenario

Solving the consensus problem is one of the most important problems in order to implement a reliable application on top of an unreliable asynchronous distributed system. In a few words the consensus problem has the purpose of reaching an agreement on common decision based on the proposed values of each node composing the network. The main problem in solving consensus in this scenario is the fact that nodes can crash, consequently is very difficult to understand when a node is really crashed or the communication channel between the nodes is congested or too “slow”.

The impossibility of solving consensus is overcome by the introduction of failure detectors as explained in [1, 6], which are modules associated to each process that give to it a list of

processes that are considered as suspected to be failed. There are many kind of failure detectors each one varying from the others for the different properties they have. The failure detectors are classified in many classes identified by two main properties: completeness and accuracy. The *completeness* property regards the number detections on the total of faulty processes, while the *accuracy* is about the quality of the suspicions, this means that the aim of the accuracy property is to reduce as possible the wrong suspicion of the failure detector regarding the non faulty nodes.

In this work we try to implement consensus with the using of different unreliable failure detectors belonging to different classes. Afterwards we will explain which kind of failure detector we have chosen in our proposed solution based on the model illustrated in [1, 6].

## 2 The consensus problem

### 2.1 Introduction

As already said the consensus problem is defined as follows: each non faulty process has to propose a value to all the other processes, then they must agree on a common value among the proposed ones. The big relevance of this problem is that many other problems can be reduced to it like *atomic broadcast* and *weak atomic commitment*. But solving consensus in the scenario described in 1, is not trivial, in fact we cannot easily identify “slow” nodes from failed nodes, moreover, as presented in [2], solving consensus tolerating even a single unannounced failure is not possible in a complete asynchronous system. So the most known solution adopted in order to overcome to this impossibility is, as described in [1, 6], the use of failure detectors. In the next sections we will give the specifications that must be followed in order to implement a correct consensus algorithm and a brief description about the different failure detectors.

### 2.2 System model

The system model considered is an asynchronous distributed model with:

- no upper bound on message delays;
- no upper bound to relative process speed;
- no synchronous clock;
- all process(nodes) know all the others;
- no communication failure, the links between the nodes are *perfect links*: they offer a reliable delivery and they not duplicate or create messages.

### 2.3 Consensus properties

In order to implement a correct consensus algorithm some specification must be followed as specified in [1, 6], in particular these specifications are the following:

- **Termination:** eventually, every correct process decides a value  $v$ ;
- **Validity:** a value  $v$  is decided by some process only if  $v$  was previously proposed by some other process;

- **Integrity:** each process decide at most once;
- **Agreement:** no two correct process decide differently.

But since the agreement property do not prevent the fact that a process can decide a value and then crash, a new property is introduced:

- **Uniform Agreement:** no two processes (correct or not) decide differently.

A consensus protocol that uses the Uniform Agreement is called Uniform Consensus and we are interested in this form.

## 2.4 Consensus protocol

Our proposed solution to solve the consensus problem is based on a rotating coordinator system. As reported in Figure 1 the working principle is very simple and follows mainly two phases. Each process has a local variable in which is saved the value of the current estimation of the decision (initially it is set to value that the node want to propose). Then the procedure is based on the concept of rounds, and at each round one coordinator is chosen between all the processes that take part in the protocol and this one manages all the operations.

A round is formed by two phases, in the first the coordinator sends its estimation value to all the other processes and it ends when either the other processes receive the value from the coordinator or when the current coordinator is suspected by the local failure detector. When a process receive a value from the coordinator it stores it in a variable otherwise, if it does not receive the value or the local *FD*(failure detector) suspects the coordinator, it stores a default value (i.e.  $\perp$ ). Therefore the aim of this phase is just try to have all the processes with the estimation variable set either with the coordinator value or with the default value.

In the second phase all the nodes share the value stored with the others. The estimation variable can be set differently from process and process due the fact that a *FD* can give wrong suspicious to the process that consequently set the estimation variable to  $\perp$ . The goal of the second phase is to ensure the agreement property: this is ensure from the fact that each process broadcasts its estimation value to all the other processes, doing that if a process receives enough identical values from the other processes it can decide for the value  $v$ , so if it does it in the round  $r$ , the nodes that proceed in the next round must do it with the estimation value set to the decided value of the node that has decided. In this manner the agreement property is never violated. The number of answers with the same value necessary to take the decision expressed before depends on the kind of failure detector that is used in the algorithm, next we will see in our implementation how this number changes.

## 2.5 Failure Detectors

In an asynchronous distributed system where there is no bound on the message delay or on process execution speed and where each process can fail we need to introduce some external modules that help us to understand when a process is crashed or not. This external modules are the *failure detectors*, oracles that gives informations on the status of the processes. A failure detector as illustrated in Figure 2 has the purpose of return a list of processes that are considered to be suspected of crash. It can make mistakes, such as it can suspect processes that are not really crashed or do not suspect nodes that are failed, furthermore it can change opinion about a process: in fact processes that was considered faulty in a given time could be then considered

no more suspected. Moreover a failure detector can give to different processes different hints about the status of a process. As we have already argued failure detectors belongs to different classes that are determined by two properties: the accuracy and the completeness.

### 2.5.1 Completeness property

Completeness is about the ability of a failure detector to suspect processes that actually crash. Completeness property is subdivided into two sub-properties: the strong completeness and the weak completeness.

- **Strong Completeness:** every process that crashes is permanently suspected by *every correct* process.
- **Weak Completeness:** every process that crashes is permanently suspected by *some correct* process.

Weak completeness are introduced since in some cases we do not want that every process tests if every other process is crashed or not.

### 2.5.2 Accuracy property

The accuracy property instead try to reduce the number of correct processes that are suspected. In this case we have four kind of accuracy:

- **Strong Accuracy:** a process is *never* suspected before it crashes by *all correct* process.
- **Weak Accuracy:** *some correct* process is *never* suspected by any correct process.
- **Eventually Strong Accuracy:** *after a time*, *all correct* processes do not suspect correct processes
- **Eventually Weak Accuracy:** *after a time*, *some correct* process is not suspected by any correct process

The first two properties are called perpetual accuracy properties since they hold all the time, while the other two are valid only after a time.

### 2.5.3 Failure Detectors Classes

With the composition of the previous properties we get eight different failure detector classes that we can see in Table 1.

	<b>Strong Acc.</b>	<b>WeakAcc.</b>	<b>Ev. Strong Acc.</b>	<b>Ev. Weak Acc.</b>
<b>Strong Compl.</b>	Perfect P	Strong S	Ev. Perfect $\diamond$ P	Ev. Strong $\diamond$ S
<b>Weak Compl.</b>	Q	Weak W	$\diamond$ Q	Ev. Weak $\diamond$ W

Table 1: Failure detectors classes

Another important thing to say is that we can reduce a failure detector in class D to another in class D' if there exists a reduction algorithm that transform the failure detector in class D to

another one in class  $D'$ . Then we can state that  $D'$  is weaker than  $D$  written  $D' \sqsubseteq D$ . Furthermore  $\sqsubseteq$  is a transitive relation such as if  $D'' \sqsubseteq D' \sqsubseteq D$  then  $D'' \sqsubseteq D$ .

### 3 Implementation

We have implemented two different version of the consensus protocol: one in which we make use of a strong failure detector and one in which we use an eventually strong failure detector. The implementations are based on the base models offered by the PeerSim framework like the event driven [3] and the cycle driven [4] models. We decide to adopt this solution because the structure of the consensus algorithm that we chose to implement is based on rounds, hence in this situation the cycle based model is particularly suitable. The event driven model instead is useful to manage the communication through the various protocol messages that arrives to the nodes.

In Figure 3 we can note the structure of a peer: each peer that implements the consensus protocol has a local module, the failure detector, which is used to obtain the information about the operational status of the other nodes of the network. In the next sections we will see the implementation details about consensus module that we have implemented. The message exchange among the nodes is done using the best effort broadcast module, that is built on the top of a reliable channel.

#### 3.1 Structure of the code

In Figure 6 there is a representation of the main classes that compose our developed solution. As we can easily see from the diagram the most important classes and core of the system are the two implementation of the consensus protocol: *ConsensusStrongFD* and *ConsensusEventuallyStrongFD*, these classes implement a common interface called *Consensus* plus all the interfaces needed by the framework to run. Associated at each consensus version there is the correspondent failure detector implementation, respectively *StrongFD* and *EventuallyStrongFD*, they implements the common interface *FailureDetector*. We define a special class to collect and share the failure detector configuration parameters called *FDParameters*. Another important class that there isn't directly correlated with the execution of the protocol but it is very useful to collect and evaluate the data produced running the simulations. In the next sections we will examine in more details some of the main components created to implement, support and run the consensus protocol.

#### 3.2 Messages

As said before the messages are the events associated to the event driven model; in our protocol we have two kind of messages, both extends the *GenericMessage* class as represented in the class diagram in Figure 7, which implements the interface *Message*. In particular the messages are the *ConsensusMessage*, which are messages used to manage the consensus protocol, and the *BEBMessage* used to manage the broadcast of the messages. A summarizing picture is shown in Figure 5 in which we also understand how the messages are encapsulated. In fact it is visible that the consensus message created at the consensus protocol level, the highest in our protocol stack, is encapsulated into the *BEBMessage* and then sent trough a reliable channel to all the nodes in the network as shown in Figure 4. Obviously when the message reach the best effort module of the other node, this is decapsulated and sent to the upper layer, the consensus one.

### 3.3 FDs implementation

In order to see the different behaviours of the consensus problem with the use of different failure detectors we decide to implement two consensus protocols each one using a different failure detector as you can see from Figure 6. The choice of the FD to use ends up in the strong and eventually strong failure detector (see section 2.5 for the theoretical details) represented by the classes *StrongFD* and *EventuallyStrongFD*, they implements the common interface *Failure-Detector*. Since the content of the work is the implementation of a consensus algorithm, and given the fact that build a failure detector is not a simple task we decided to create a module that simulates them. The simulation of a FD is quite simple since the only thing to do is to force the oracle to give correct or wrong suspicious about the operational status of the nodes basing on the kind of FD that we want to implement. In our case, where we decided to use the strong end eventually strong FD we have to consider in both cases the strong completeness characteristic and, in the first case the weak accuracy while in the second the eventually weak accuracy, as we can see in Table 1.

#### 3.3.1 Strong Failure Detector

In order to simulate a strong FD is necessary to follow the strong completeness and the weak accuracy characteristics, we will see in this section how we implement these characteristics in class classes *StrongFD*.

Considering the strong completeness property:

- every process that crashes is permanently suspected by *every correct* process.

we can deduce that is sufficient that each correct process suspects each process that is failed; therefore in our case we simply look at the status of the nodes, if they are down our FD suspects them without making any mistake.

Regarding the weak accuracy, instead, looking at the definition:

- *some correct* process is *never* suspected by any correct process.

we can note that the FD here can make mistakes in the analysis of the status of the nodes. Hence our FD given a node that is up, with a certain probability it decides if return the correct or the wrong answer. But we have to pay attention that some answers returned by the FD must be correct (note the *some* correct process is never suspected ...). Therefore we limit the possible wrong suspicious of the FD imposing to the FD to give at least one correct hint.

We can see in the listing 1 how we preserve the characteristics argued above introducing the possibility of mistake on node detection.

Listing 1: StrongFD.java

```
public ArrayList<Long> callOracle() {  
  
    double prob = 0;  
    countStartTossing = 0;  
  
    logger.info("***** ORACLE CALLED AT NODE "  
               + CommonState.getNode().getID() + "  
               *****");  
  
    for (int i = 0; i < Network.size(); i++) {
```

```

if (Network.get(i).isUp()
    && (Network.get(i).getID() != CommonState.getNode
        ().getID())) {

    // node up so FD can take a wrong decision on it

    // we skip the node to not suspected
    // we subtract to the network size the number of failed
    // nodes
    // and the number of nodes that may be wrong suspected
    // if we potentially suspected more than network size – failed
    // – 1
    // then we set the potentially suspected to network size –
    // failed – 1
    // in order to obtain a StrongFD (at least one node is never
    // wrongly suspected) because failed node are already
    // suspected

    if (nPossibleWrongSuspected < 0)
        nPossibleWrongSuspected = 0;
    if (nPossibleWrongSuspected > Network.size() –
        fdParameters.getFailed())
        nPossibleWrongSuspected = Network.size() –
            fdParameters.getFailed()
                – 1;

    // counter for end tossing the coin
    // stops when reach nPossibleWrongSuspected value
    // counterChangeHint++;

    countStartTossing++;

    logger.info("node " + Network.get(i).getID() + " is up");

    logger.info("At Node " + CommonState.getNode().getID()
        + ": possible wrong suspected = "
        + nPossibleWrongSuspected + ", local
        counter of up nodes = "
        + countStartTossing);

    if (countStartTossing <= nPossibleWrongSuspected) {

        prob = (Math.random() * 1);

        if (prob > fdParameters.getProb()) {
            logger.info("At Node " + CommonState.
                getNode().getID()
                + ": tossed value = " + prob
                + ", threshold value to be
                correct = "
                + fdParameters.getProb() + "
                The NODE "
                + Network.get(i).getID() + "

```

```

        is CORRECT!");
        if (isSuspected(Network.get(i)))
            removeSuspected(Network.get(i));
    } else {
        logger.info("At Node " + CommonState.
            getNode().getID()
                + ": tossed value = " + prob
                + ", threshold value to be
                correct = "
                + fdParameters.getProb() + "
                The NODE "
                + Network.get(i).getID() + "
                is SUSPECTED!");
        addSuspected(Network.get(i));
    }
}
} else {
    if (Network.get(i).getID() != CommonState.getNode().getID())
    {
        addSuspected(Network.get(i));
    }
}
}

logger.info("***** RESULT OF ORACLE
*****");
logger.info(toString());
logger.info("***** END ORACLE
*****");

return suspected;
}

```

### 3.3.2 Eventually Strong Failure Detector

In the case of the eventually strong failure detector implemented in class *EventuallyStrongFD* we have to change a little bit only the implementation of the accuracy property, in fact the completeness property is like the case of the strong FD; looking at the definition we get that:

- *after a time, some correct process is not suspected by any correct process.*

Therefore in this case the hints can be wrong for all the nodes, but only for a certain time; when this period ends some correct process is no more suspected; hence we decide to allow the FD to make mistake for all the node for a given time, after that at least one correct node will not wrongly suspected any more. This means that if before the limit it was suspected then it will be removed from the suspected list.

We can see in the listing 1 how we preserve the characteristics argued above introducing the possibility of mistake on node detection.

Listing 2: EventuallyStrongFD.java

```

public ArrayList<Long> callOracle() {

    double prob = 0;

    if (CommonState.getTime() > fdParameters.getTimeThreshold()) {

        logger.info("##### STOP TOSSING COIN AT
        NODE "
            + CommonState.getNode().getID() + "
            #####");

        // we skip the node to not suspected
        // we control potentiallySuspectedAfter and in the case too bigger
        // we fix it to network.size - failed - 1

        logger.info("potentiallySuspectedAfter = "
            + potentiallySuspectedAfter);

        if (potentiallySuspectedAfter < 0)
            potentiallySuspectedAfter = 0;
        if (potentiallySuspectedAfter > Network.size()
            - fdParameters.getFailed())
            potentiallySuspectedAfter = Network.size()
            - fdParameters.getFailed() - 1;

        // Control the number of suspected node not exceed
        // potentiallySuspectedAfter

        for (int c = 0; c < Network.size(); c++) {

            if (Network.get(c).isUp()) {
                if ((suspected.size() > potentiallySuspectedAfter
                    && isSuspected(Network.get(c))) {
                    removeSuspected(Network.get(c));
                    logger.info("Remove from suspected node "
                        + Network.get(c).getID());
                }
            }
        }

        logger.info("##### ORACLE HINTS NOT
        CHANGE ANYMORE AT NODE "
            + CommonState.getNode().getID() + "
            #####");

        logger.info("***** RESULT OF ORACLE AT NODE "
            + CommonState.getNode().getID() + "
            *****");
        logger.info(toString());
        logger.info("***** END ORACLE AT NODE "
            + CommonState.getNode().getID() + "
            *****");

        return suspected;
    }
}

```

```

}

logger.info("***** ORACLE CALLED AT NODE "
           + CommonState.getNode().getID() + "
           *****");

logger.info("fdParameters.getProb() of the hint to be correct: "
           + fdParameters.getProb());

for (int i = 0; i < Network.size(); i++) {
    if (Network.get(i).isUp()
        && (Network.get(i).getID() != CommonState.getNode()
            .getID())) {
        // node up so FD can take a wrong decision on it

        prob = (Math.random() * 1);

        logger.info("node " + Network.get(i).getID() + " is up");

        if (prob > fdParameters.getProb()) {
            logger.info("At Node " + CommonState.getNode().
                getID()
                + ": tossed value = " + prob
                + ", threshold value to be correct = "
                + fdParameters.getProb() + " The
                NODE "
                + Network.get(i).getID() + " is
                CORRECT!");
            if (isSuspected(Network.get(i)))
                removeSuspected(Network.get(i));
        } else {
            logger.info("At Node " + CommonState.getNode().
                getID()
                + ": tossed value = " + prob
                + ", threshold value to be correct = "
                + fdParameters.getProb() + " The
                NODE "
                + Network.get(i).getID() + " is
                SUSPECTED!");
            addSuspected(Network.get(i));
        }
    } else {
        if (Network.get(i).getID() != CommonState.getNode().getID())
        {
            addSuspected(Network.get(i));
        }
    }
}

logger.info("***** RESULT OF ORACLE
           *****");

```

```

        logger.info(toString());
        logger.info("***** END ORACLE
        *****");
        return suspected;
    }

```

---

### 3.4 Consensus implementation

Starting from the same common interface *Consensus* as we can see in Figure 6, we implement two different consensus protocols: *ConsensusStrongFD* and *ConsensusEventuallyStrongFD*, since they change depending on the kind of FD used. In reality the two consensus implementations do not differ very much the one from the other; in fact the structure remains the same for both the solutions, the only thing that changes is the control about the necessary number of proposed value received that are needed in order to allow a node to take a decision. We will see below in the listings 3 and 4 this main difference. Furthermore the different solutions tolerate a different number of faulty processes. Afterwards we will see the two different implementations, the one with the strong failure detector and the one with the eventually strong failure detector.

Listing 3: ConsensuStrongFD.java

```

if ((answers.size() + suspected.size() == Network.size())) {

```

---

Listing 4: ConsensuEventuallyStrongFD.java

```

if (answers.size() > (Network.size() / 2)) {

```

---

#### 3.4.1 Consensus using strong failure detector

The usage of the strong failure detector impacts, as said before, on the number of nodes that the consensus algorithm tolerates; in this case, if we have a network of  $n$  nodes, the number of faulty nodes tolerated is  $n - 1$ ; In fact as we will see in the analysis, we can put up to  $n - 1$  nodes to the down state and our algorithm works reaching a decision. Furthermore to get a decision a node must wait to receive a value from all the nodes that are not in the list of the suspected nodes.

#### 3.4.2 Consensus using eventually strong failure detector

With the eventually strong failure detector the things change, in fact now the consensus algorithm tolerates less faulty nodes respect to the previous solution. Now the number of failed nodes tolerated is  $\lceil n/2 \rceil$ . Also the condition to get the decision changes, in this case is necessary to receive at least  $\lceil (n + 1)/2 \rceil$  values from the other nodes in order to take a decision on a value.

## 4 Simulation

In this chapter we will discuss about the elements useful to execute the simulation of our protocol. We will go into details examining the configuration files and their parameters, the initializers used to build the topology of the network or to set the initial node values and finally the

observers, a useful tool that help us to collect data on the simulation and create some statistics. We will better understand which are the parameters common to all the protocol instance and which are specific for a particular simulation.

## 4.1 Configuration files

The consensus protocol can be customize through a configuration file, in reality the configurations files are two, one for the case of the strong FD and one for the eventually strong FD. Both the configuration files has some common parameters to set up the network, the node stack, or the control structure and other particular for the failure detectors. hence, we will describe in details the two configuration files illustrating the parameters in common and the specific parameter relative to the failure detectors.

### 4.1.1 Network Parameters

The first group of common parameters are listed below: the constants useful to build the network like the size of the network, the number of failed nodes, the number of cycles and the duration of each cycle.

Listing 5: consensus\_strong.cfg

```
# Network size
SIZE 10

# Parameters of periodic execution
CYCLES 30
CYCLE SIZE * 1000

# Number of nodes in the network that fails
FAILED 5
```

---

### 4.1.2 Simulation Engine Parameters

These parameters defines the dimension of network and time constraints.

Listing 6: consensus\_strong.cfg

```
random.seed 1234567890
network.size SIZE
simulation.endtime CYCLES * CYCLE
simulation.logtime CYCLE
```

---

### 4.1.3 Protocol Parameters

These are the core parameters because they establish how is built the protocol stack inside a node as reported in Figure 3. For example in this instance the application protocol on top of the stack is the consensus with strong FD, this particular protocol use as transport protocol for the messages the Best Effort Broadcast, which in turn send the messages using as a service a reliable transport protocol.

Listing 7: consensus\_strong.cfg

```
protocol.link peersim.core.IdleProtocol

protocol.consensus it.unitn.ds.peersim.protocols.ConsensusStrongFD
protocol.consensus.linkable link
protocol.consensus.step CYCLE
protocol.consensus.transport beb

protocol.beb it.unitn.ds.peersim.protocols.BestEffortBroadcast
protocol.beb.transport rt

protocol.rt it.unitn.ds.peersim.protocols.transports.ReliableTransport
```

---

#### 4.1.4 Initialization Parameters

As mentioned above we need to initialize the simulation, to build a topology on top of it the protocol runs, to define the initial distribution of the values and to schedule which nodes start to run the protocol as first.

Listing 8: consensus\_strong.cfg

```
init.failed FailedNodes
init.failed.number FAILED
init.failed.protocol link

init.rndlink WireKOut
init.rndlink.k SIZE
init.rndlink.protocol link

init.vals UniformDistribution
init.vals.protocol consensus
init.vals.max -1.0
init.vals.min -1.0

init.sch CDScheduler
init.sch.protocol consensus
inti.sch.randstart
```

---

#### 4.1.5 Control Parameters

The last group of common parameters is the one report the control parameters. This set of parameters define which is the *Observer* class which has the job of collect simulation data and elaborate some statistics. The *step* parameter defines the polling intervall for the Observer class.

Listing 9: consensus\_strong.cfg

```
control.obs it.unitn.ds.peersim.observers.ConsensusObserver
control.obs.protocol consensus
control.obs.accuracy 100
control.obs.step CYCLE
```

---

#### 4.1.6 consensus\_strong.cfg

Once we have understand which are the common parameters relative to each simulation we examine the parameters that we can set to customize the strong failure detector:

- **PROB:** probability of the FD to give a wrong hint about the correct processes;
- **P\_WRONG\_SUSPECTED:** number of nodes that can be wrongly suspected by the FD, remember that a strong FD never suspects some correct processes, hence, the maximum number of nodes to set is  $SIZE - FAILED - 1$ .

#### 4.1.7 consensus\_ev\_strong.cfg

The parameters that we can set to customize the eventually strong failure detector:

- **PROB:** probability of the FD to give a wrong hint about the correct processes;
- **P\_WRONG\_SUSPECTED:** number of nodes that can be wrongly suspected by the FD, remember that a strong FD never suspects some correct processes, hence, the maximum number of nodes to set is  $SIZE - FAILED - 1$ ;
- **TIME\_THRESHOLD:** time after which some correct processes are no more suspected by the failure detector.

### 4.2 Initializer

The consensus module provide one initiator used to set the state of the node to down. The number of nodes to put in the down state is specified in the configuration file using the parameter `FAILED`. As all the initiator it is executed at the beginning of the simulation.

### 4.3 Observer

The observer, called `ConsensusObserver`, has the purpose of logging the entire execution of the consensus problem, in particular at each cycle are logged the number of nodes that have decided and which value; furthermore the observer stops the execution when it reaches the sufficient number of decided nodes. All the logs created by this observer are then used for the generation of the graphs.

### 4.4 Running a simulation

To run a simulation is enough to launch the following command:

```
java -cp "peersim-1.0.jar:jep-2.3.0.jar:djep-1.0.0.jar" peersim.Simulator cfg/consensus_strong.cfg
```

To execute instead the consensus protocol with the eventually strong FD is sufficient to change the configuration file to `cfg/consensus_ev_strong.cfg`

### 4.5 Running multiple simulations

If you want to run multiple simulation, you can use the specific script we create to run a test multiple times. This is a shell scrip which takes as parameter the number of test decided. Hence, to launch a hundred test, you type on the shell: `sh run.sh 100`

Below you see the content of the script to better understand what happen.

Listing 10: run.sh

```
#!/bin/bash

# USAGE sh run.sh test_number

#####
# Distributed Systems Course Project
# Year: 2009/2010
# University of Trento
#
# Author: Avancini Mattia
# Author: Giampaolo Farina
# Version: 1.0
#####

if [ $# -ne 1 ]; then
    echo "USAGE: sh run.sh test_number"
    exit
fi

test_number = $1
test_number = test_number + 1

rm -rf log

echo "-----> START running experiments"
counter=1          # Initialise a counter
while [ $counter -lt test_number ]      # Set up a loop control
do
    echo "-----> START running experiment: $counter"
    java -cp "./jar/DSPProject.jar" peersim.Simulator cfg/consensus_strong.cfg
    echo "-----> END running experiment:"
    counter=$(( $counter + 1 ))          # Increment the counter
done
echo "-----> END running experiments"
exit
```

## 5 Analysis and results

In this section we will present the results of our analysis, in particular we have tested our algorithms with a network of 10 nodes in order to better understand the behaviour of our protocols. In fact if we use more nodes the graphs analysis would be more difficult and not useful for our purposes. Hence, starting from the network with 10 nodes we execute our algorithms varying the parameters explained in sections 4.1.6 and 4.1.7.

In the case of the protocol using the strong failure detector we execute 100 tests for each parameters setting; in particular we made the tests varying the number of failed nodes, we started from one failed node and we increased the number progressively until we reach eight failed nodes. We did not go on since we set the *P\_WRONG\_SUSPECTED* parameter to 2, so potentially we could get ten faulty nodes that means the entire network. Regarding the probability of our failure detector to make mistakes we set it to 0.1 setting the parameter *PROB*.

Also in the case of consensus with eventually strong failure detector we made 100 tests for each different parameters setting. We set the *PROB* parameter to 0.1 while the *TIME\_THRESHOLD* parameter to 100. Again we started from one failed node and increased the value until 5 (Note that with the eventually strong failure detector the limit of tolerated failed nodes is  $\lceil n/2 \rceil$ ).

## 5.1 Graphs generation

The graphs generation is performed with the use of the gnuplot tool [7] and the procedure is fully automated; we created some templates and starting from them and the data collected in the logs during the execution of the module, we generated all the plots by the using of some bash scripts. The main script is the *DS\_get\_data.sh* that moves the logs contained in the log folder of the java project and creates the folders needed to store the results. This script is based on the execution of the sub-scripts:

*DS\_renamer.sh*: renames the log files adding an index that indicates the number of test that generates it.

After that other two bash scripts compute the necessary informations and create the plots:

*DS\_get\_consensus\_time.sh*: creates a file, for each test set, containing the time of the cycle in which each test of the set reaches the consensus. Then it computes the average time of each test set and store the results in a file called *avg\_consensus\_time.log*

*DS\_graphs.sh*: generates all the graphs starting from the templates present in the *gnu\_template* folder.

To run the script *DS\_get\_data.sh* is necessary to type:

```
bash DS_get_data.sh data_folder StrongFD
```

where *data\_folder* is the folder where to store the data files and the resulting graphs, *StrongFD* is the name of the failure detector used to generates the data files. In the case the eventually strong FD is used the parameter passed must be *EvStrongFD*.

While to run the other two scripts is necessary to type:

```
bash DS_get_consensus_time.sh data_folder StrongFD
```

```
bash DS_graphs.sh data_folder StrongFD
```

where the *data\_folder* is the folder where the data was stored and the last parameter works as for the previous script.

## 5.2 Analysis and comments

At the end we got two kinds of graphs: the first is a summarising graph that shows the cycle in which each of the sets of tests reaches the consensus, an example can be seen in Figure 8, the second instead is a specific test graph that shows the evolution of the test, such as it shows for each node in the graph the nodes that it suspected, differentiating the failed ones from the nodes that are wrongly suspected, the cycle in which the nodes decides, and since the proposed value of each node is its id, also the value decided. Furthermore the failed nodes are highlighted by the using of an horizontal line, so that their identification is more intuitive. An example of this kind of graph is shown in Figure 9.

### 5.3 Consensus with strong failure detector

In Figure 8 is shown the consensus time graph about the tests done, we made a 8 sets of 100 tests, in each set we increased the number of failed nodes, starting from 1 to 8. Observing it we can note that when the number of failed nodes is low the consensus time is for the major part of the tests in the same cycle, when the number of failed nodes increase the consensus time is spread in many cycles.

In Figure 9, 10 and 11 there are three examples of the evolution of the protocol. In particular we can note that in Figure 11 the consensus is reached in the second cycle since the coordinator is failed.

Taking the tests in which eight nodes are failed we note that the time needed to get the consensus is higher, this is easily visible looking at the Figure 8. From the Figure 30, 31 and 32, instead, we can see that in some cases the consensus is reached while in some others not; specifically in Figure 32 we note that the only two non faulty nodes are suspected each other by their local failure detectors, therefore the consensus cannot be reached. In Figure 30 is represented the case in which the consensus is reached, since none of the two non faulty nodes are never suspected by theirs failure detectors.

In appendix A.1 there are shown the graphs about all the other tests.

### 5.4 Consensus with eventually strong failure detector

In the case of consensus with eventually strong failure detector we note that the time in which the consensus is reached is more spread respect to consensus with strong failure detector as we can see in Figure 33. The cause of this difference is that with the eventually strong FD all the correct nodes in the network are potentially continuously suspected for an initial period of time. This means that the coordinator in a round can be wrongly suspected by many nodes, this can bring the protocol to pass to the next round choosing a new coordinator and the consequence is the increasing of the time needed to reach the consensus.

Furthermore as already said the number of tolerated failed nodes are less than consensus with strong FD.

All the graphs about consensus with eventually strong failure detector can be seen in appendix A.2.

## 6 Conclusion

At the end with this paper we have described the theory behind the consensus problem and the failure detectors. In particular we have focused our attention on the failure detectors since they are essential in order to solve consensus in an unreliable distributed system. After that we have given two possible implementations of the consensus protocol one using the strong failure detector and one using the eventually strong one using the Peersim engine. Then we have made several simulations using both the implementations, and varying the various parameters that characterise the failure detectors getting the expected results, such as we have shown that our algorithms works as explained by the theory. In fact with the strong failure detector we can have a number of failed nodes equal to the size of the network minus one, while in the case of the eventually strong failure detector we can have a number of failed nodes equal to  $\lceil n/2 \rceil$ .

# A Figures

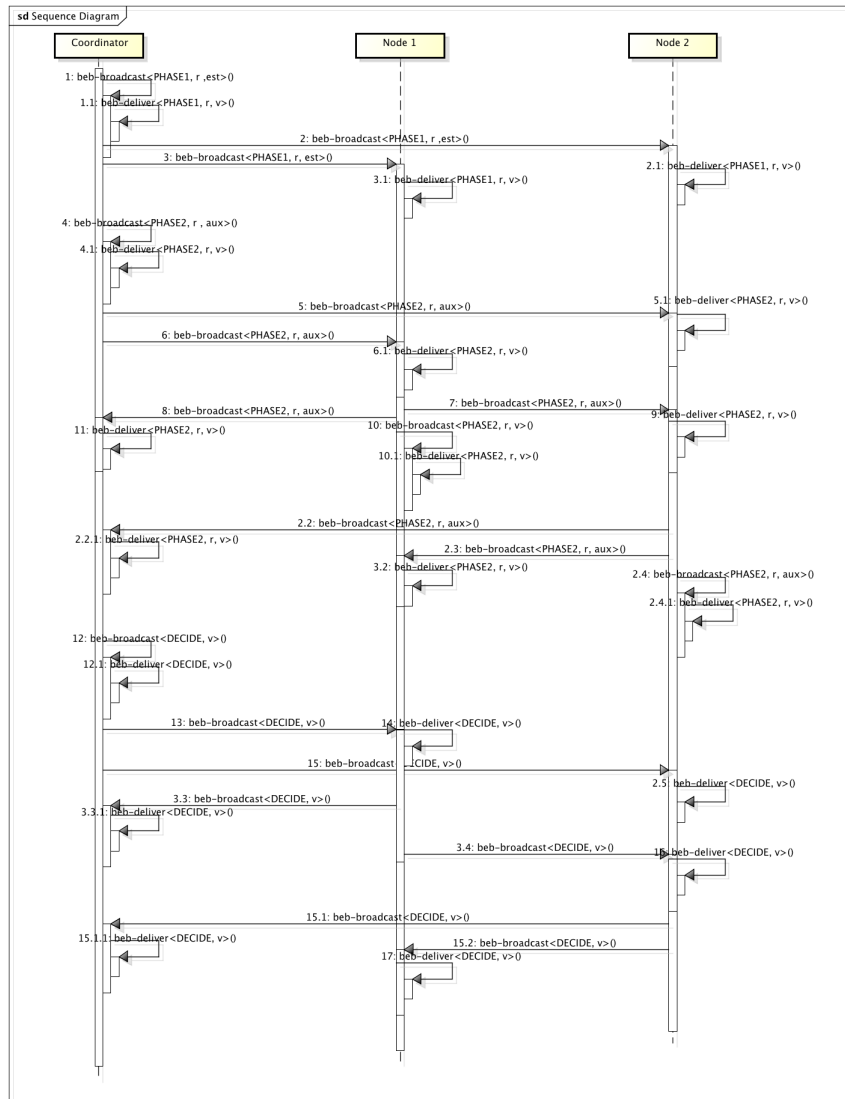


Figure 1: Sequence Diagram of a possible instance of consensus protocol

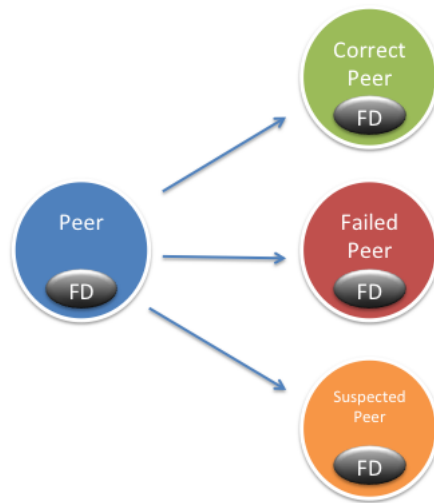


Figure 2: Failure detector module

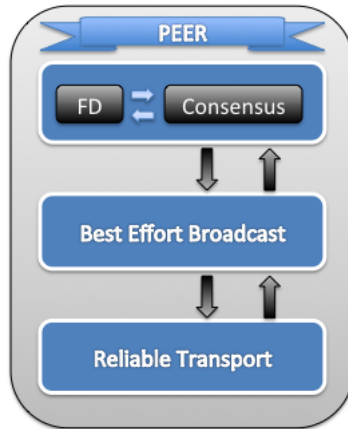


Figure 3: Peer structure

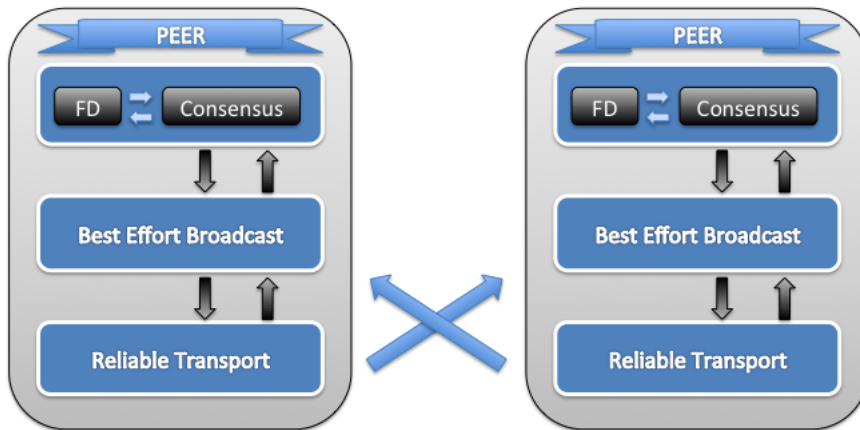


Figure 4: Network architecture

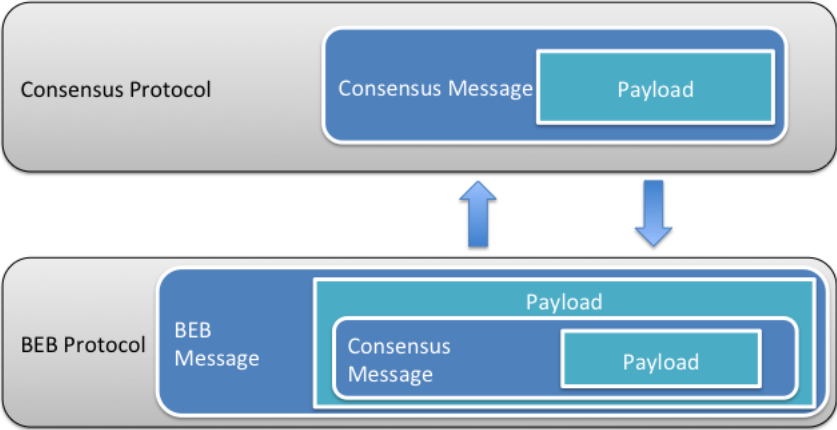


Figure 5: Messages structure

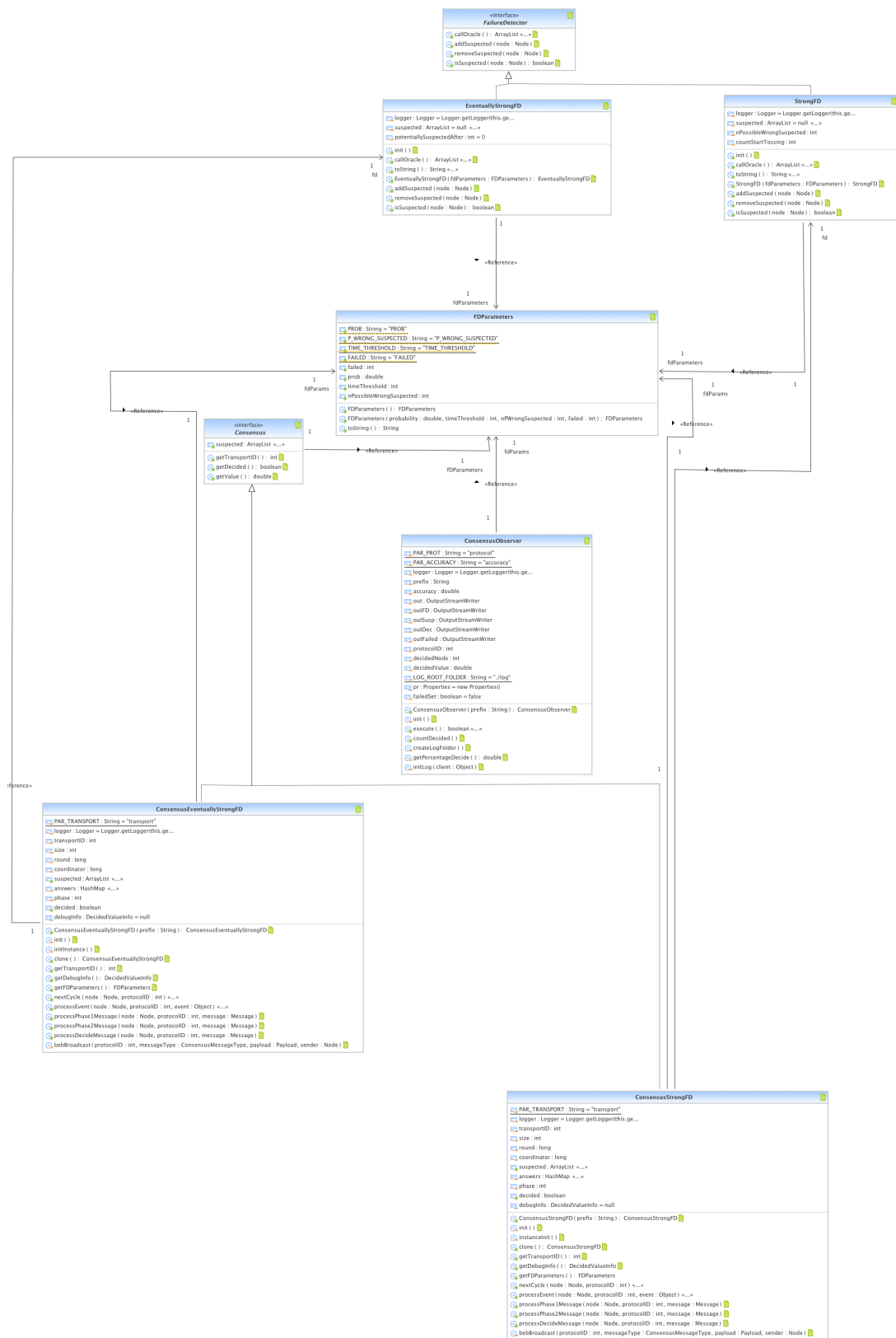


Figure 6: Class Diagram Main Classes

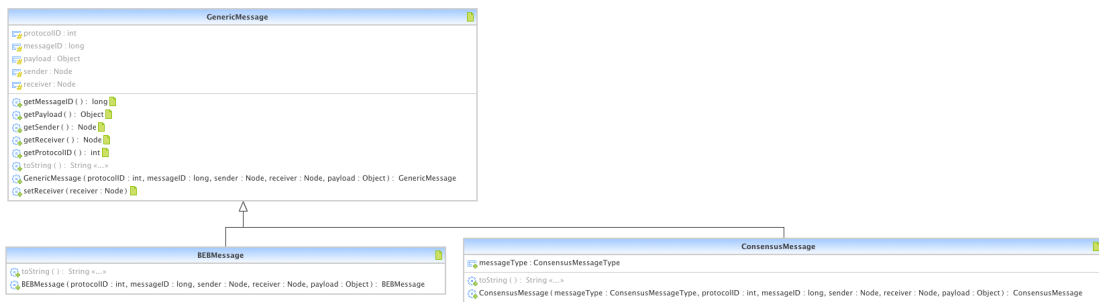


Figure 7: Class Diagram Messages

## **A.1 Consensus with Strong FD**

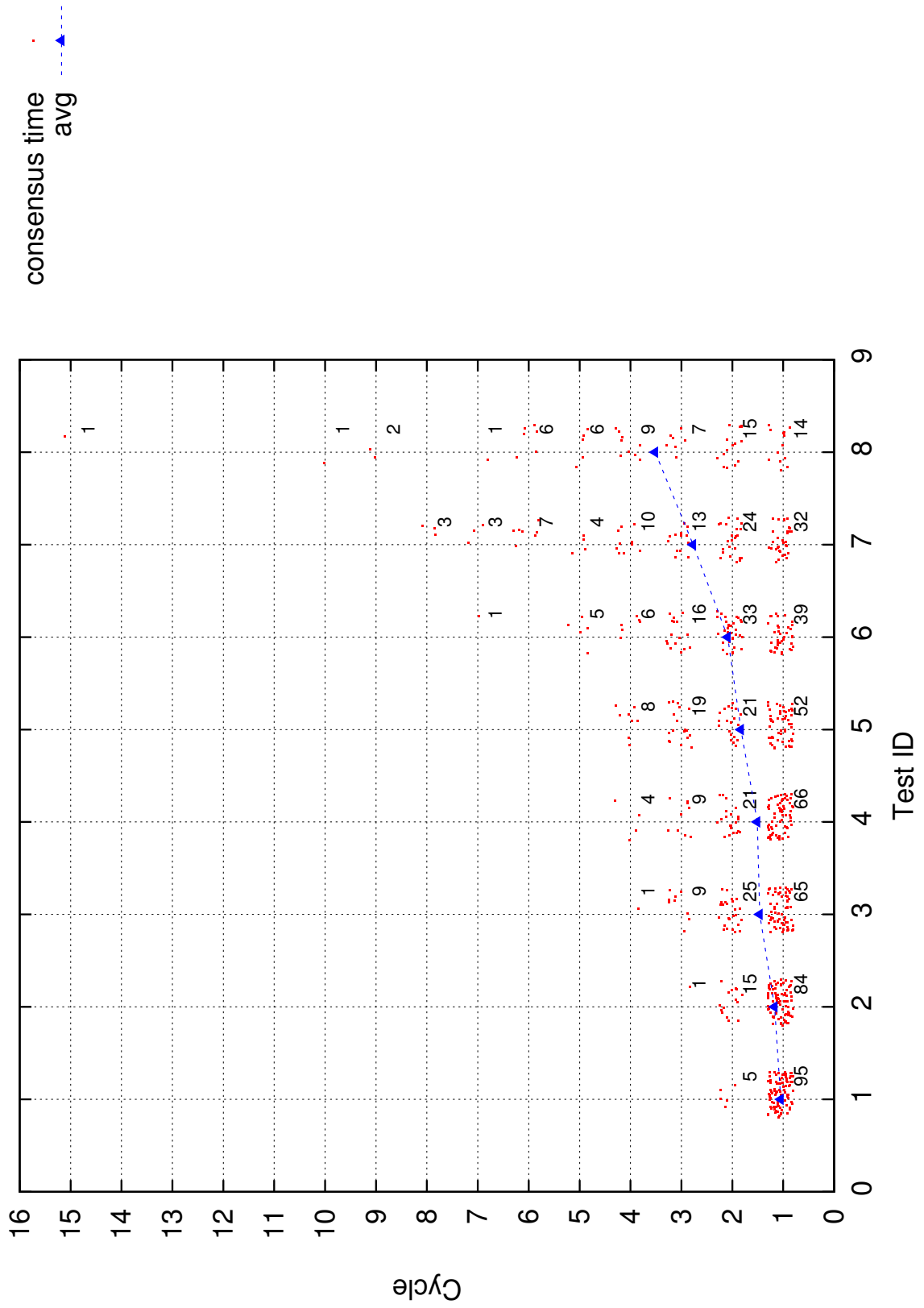


Figure 8: Consensus times

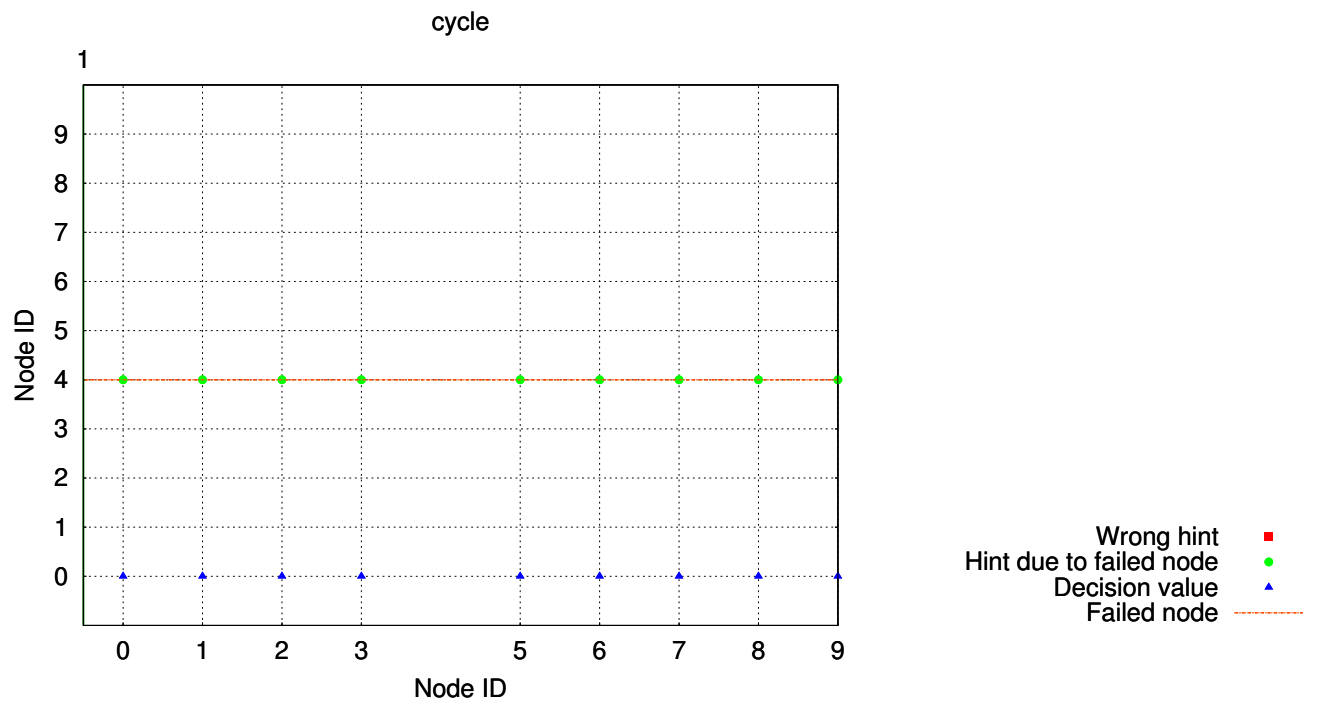


Figure 9: Example of test with 1 failed node, summarizing picture about the nodes status

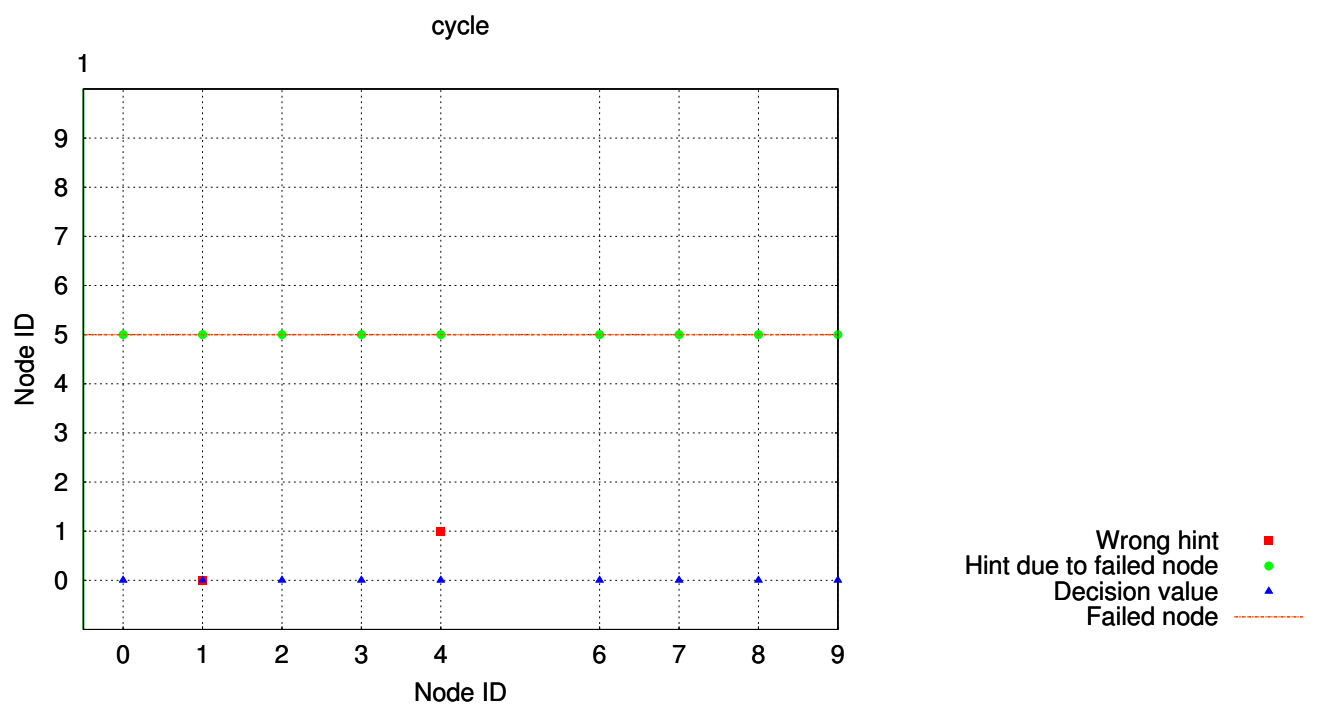


Figure 10: Example of test with 1 failed node, summarizing picture about the nodes status

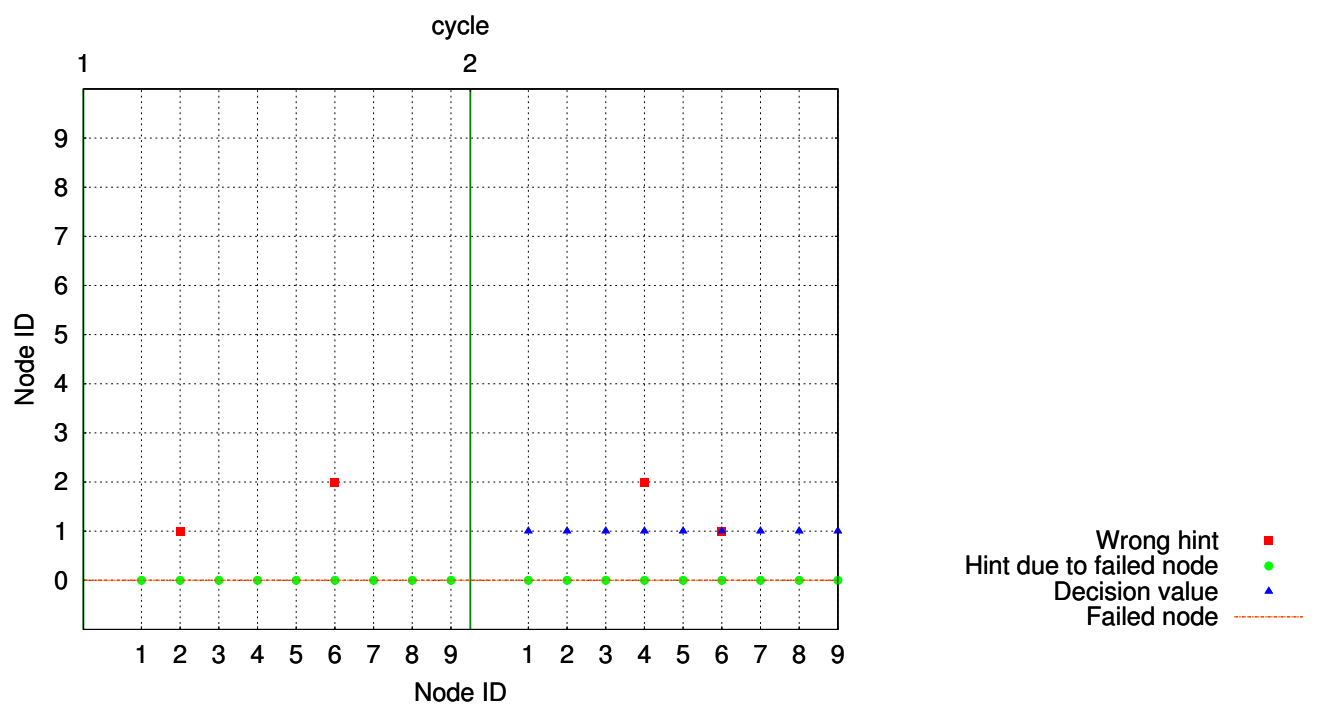


Figure 11: Example of test with 1 failed node, summarizing picture about the nodes status

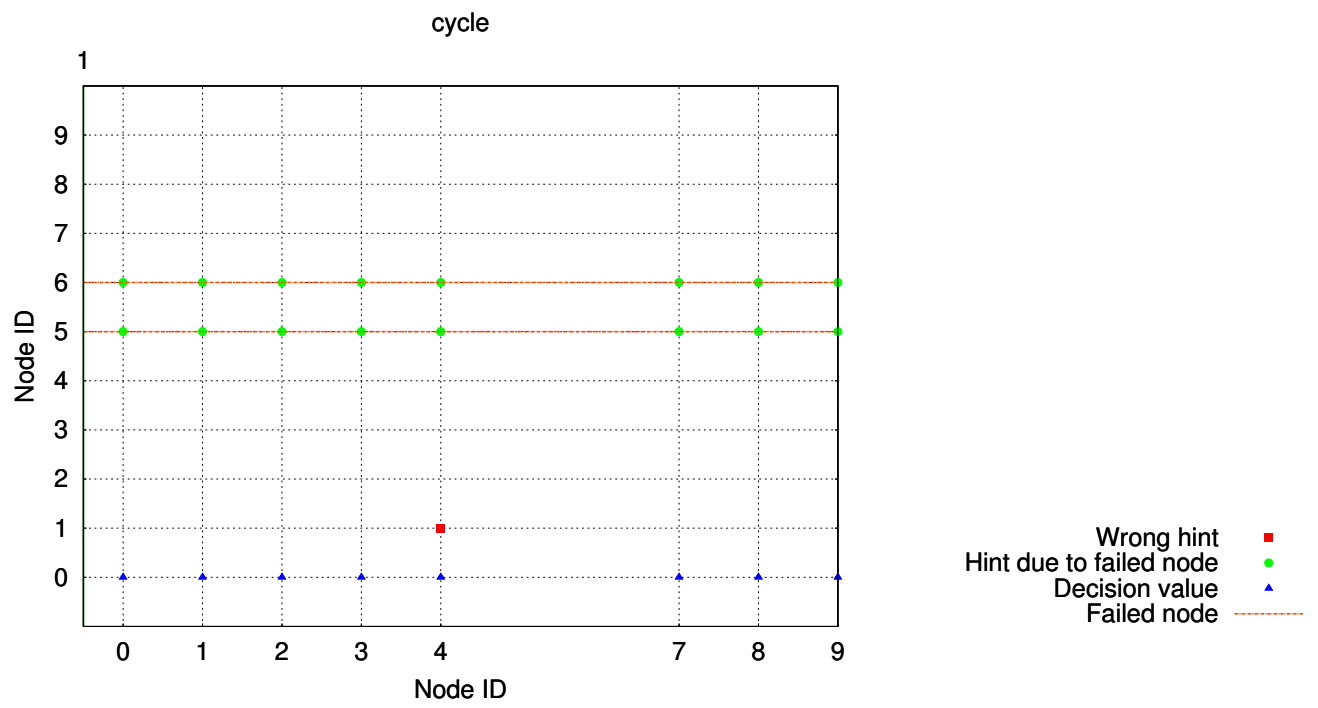


Figure 12: Example of test with 2 failed nodes, summarizing picture about the nodes status

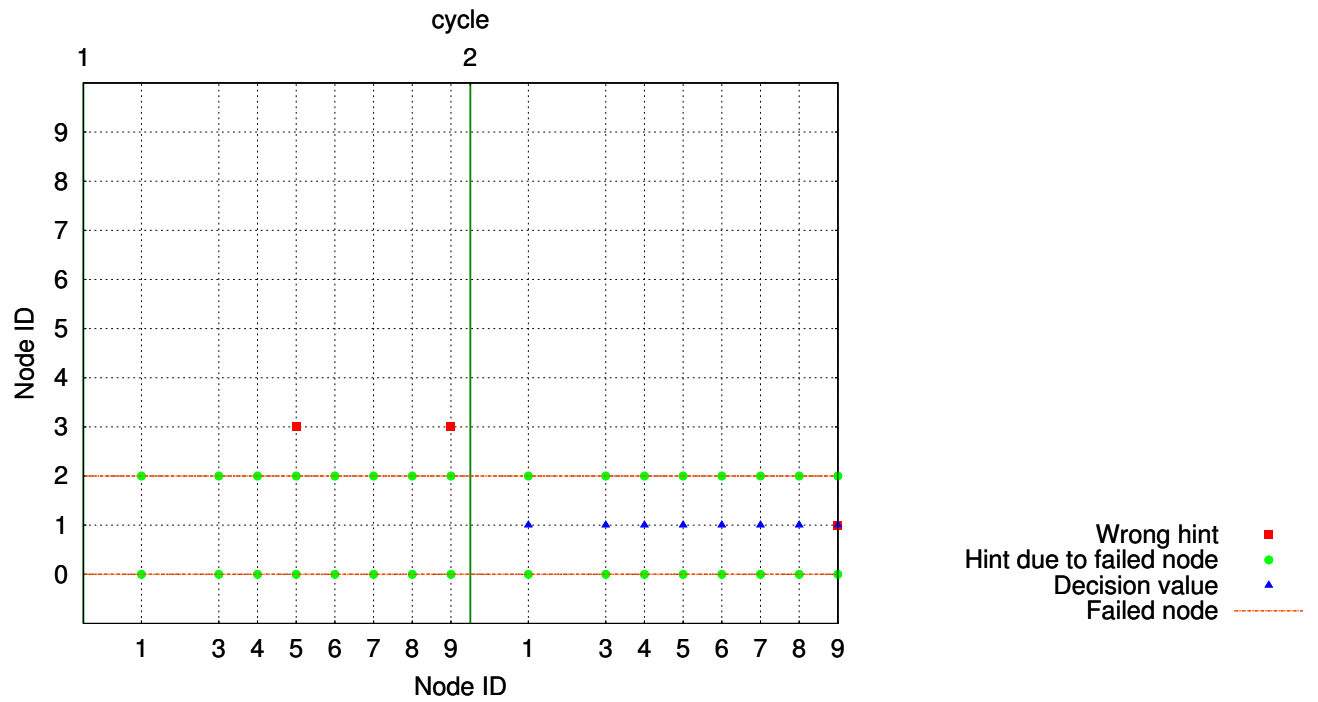


Figure 13: Example of test with 2 failed nodes, summarizing picture about the nodes status

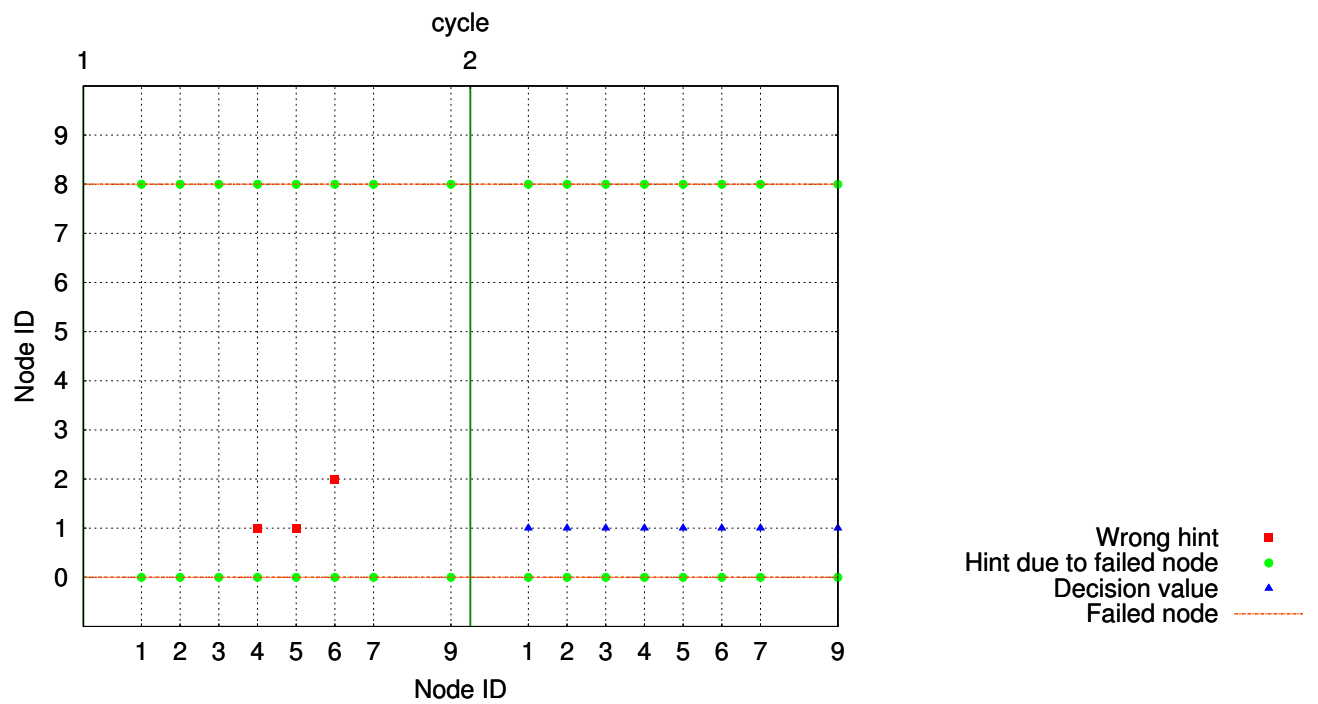


Figure 14: Example of test with 2 failed nodes, summarizing picture about the nodes status

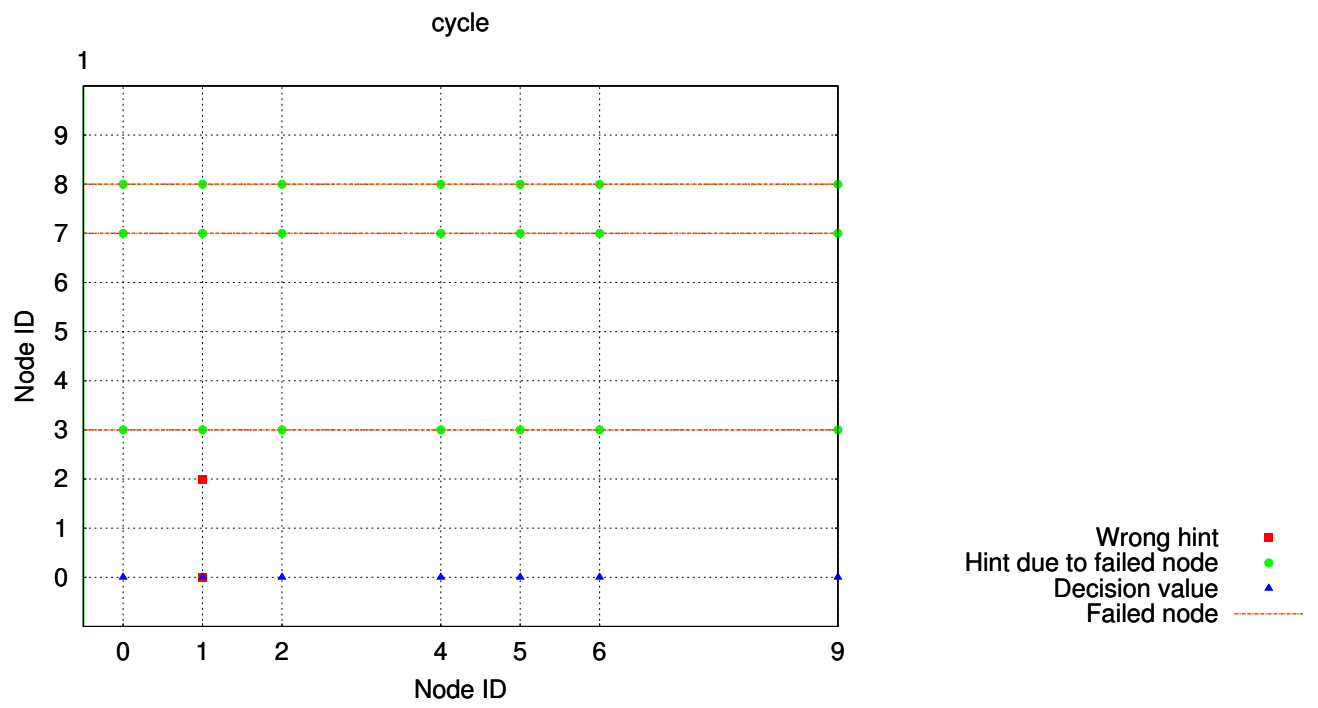


Figure 15: Example of test with 3 failed nodes, summarizing picture about the nodes status

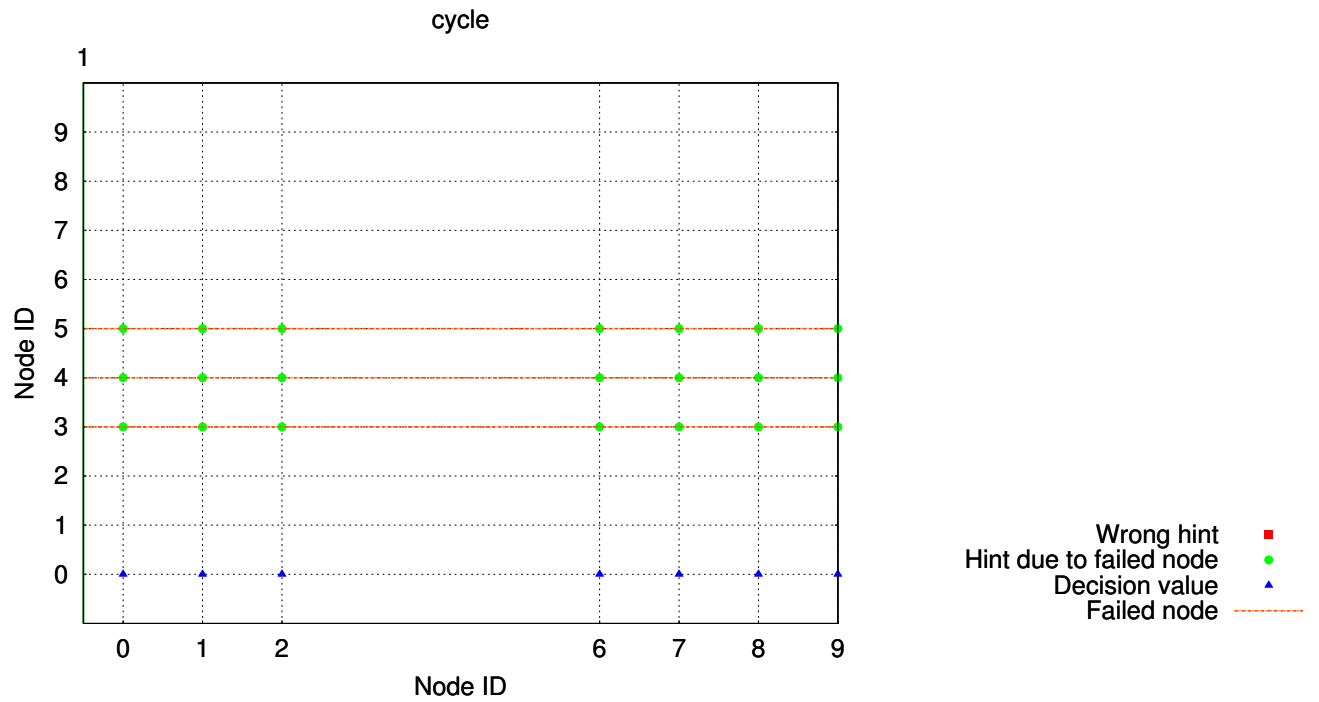


Figure 16: Example of test with 3 failed nodes, summarizing picture about the nodes status

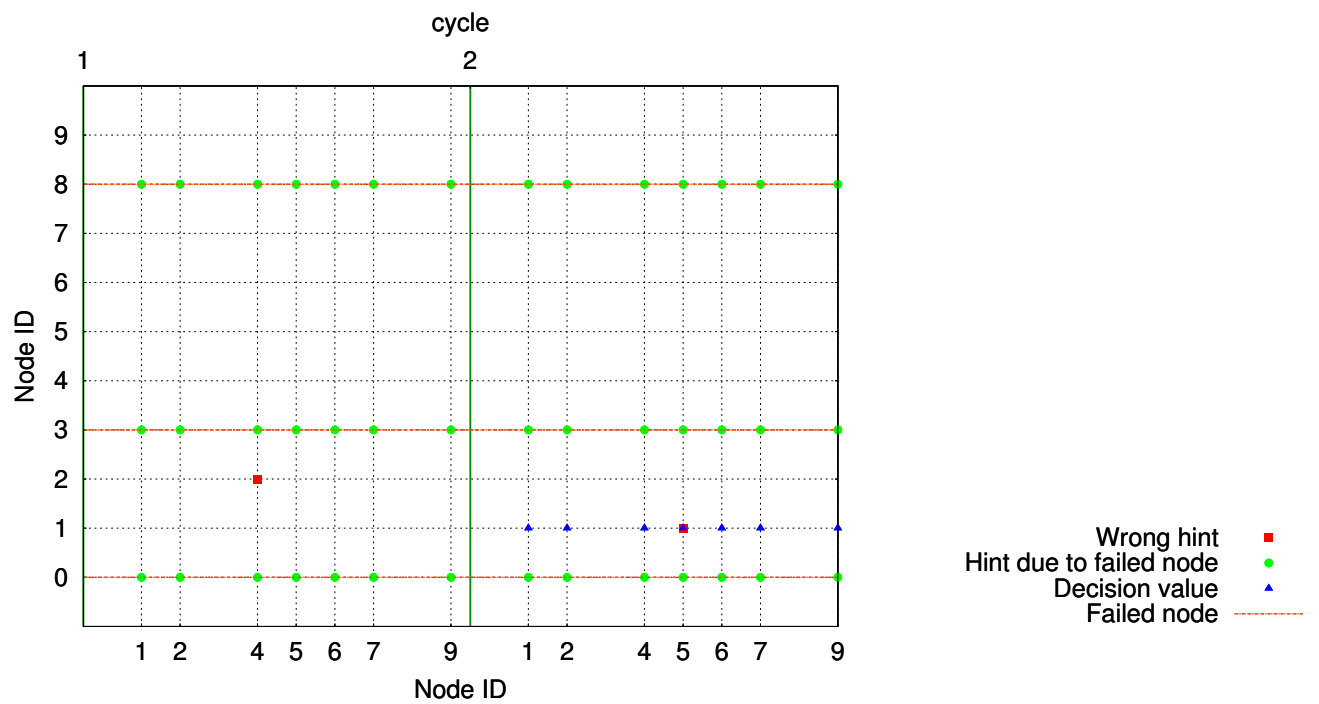


Figure 17: Example of test with 3 failed nodes, summarizing picture about the nodes status

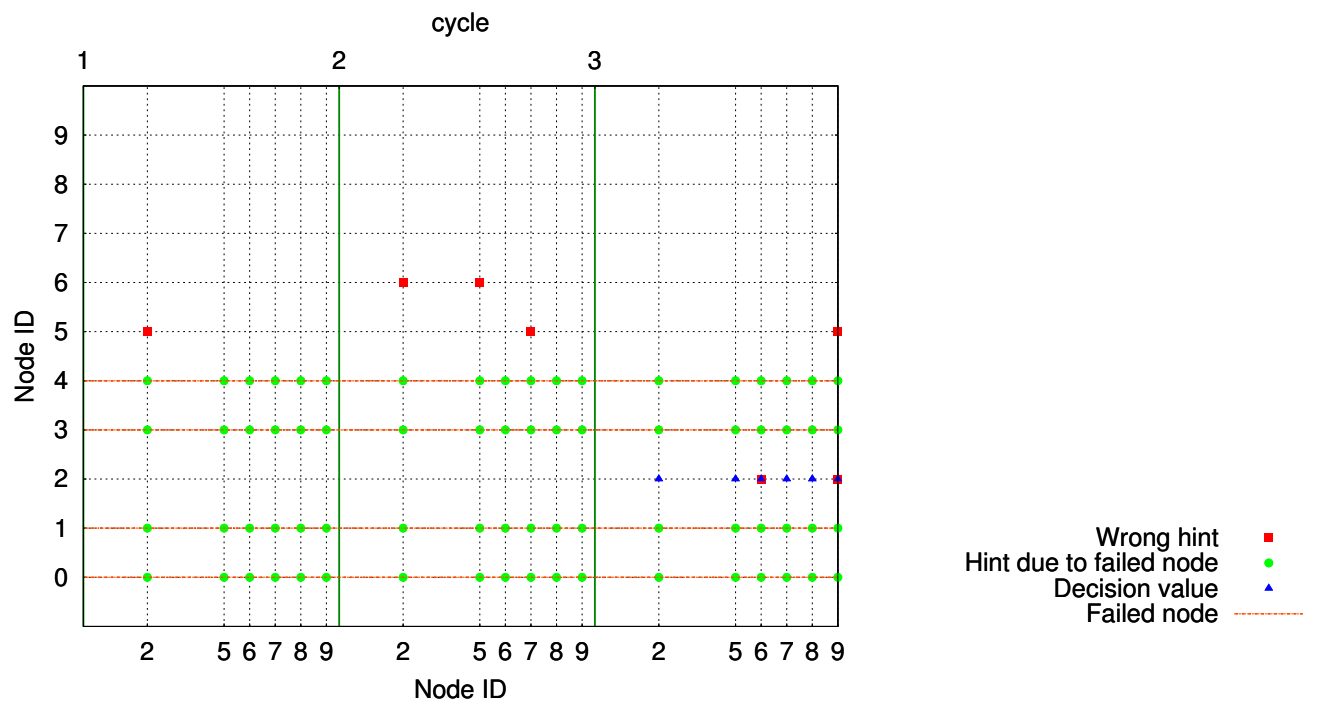


Figure 18: Example of test with 4 failed nodes, summarizing picture about the nodes status

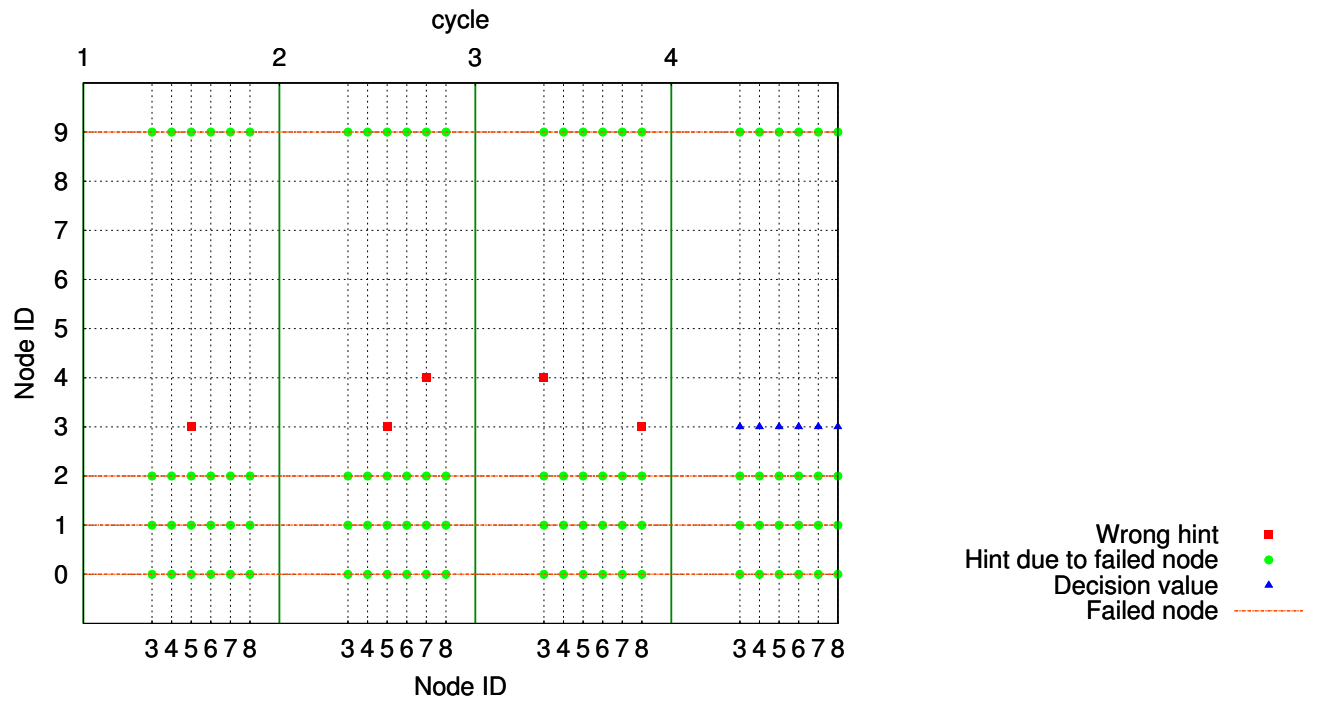


Figure 19: Example of test with 4 failed nodes, summarizing picture about the nodes status

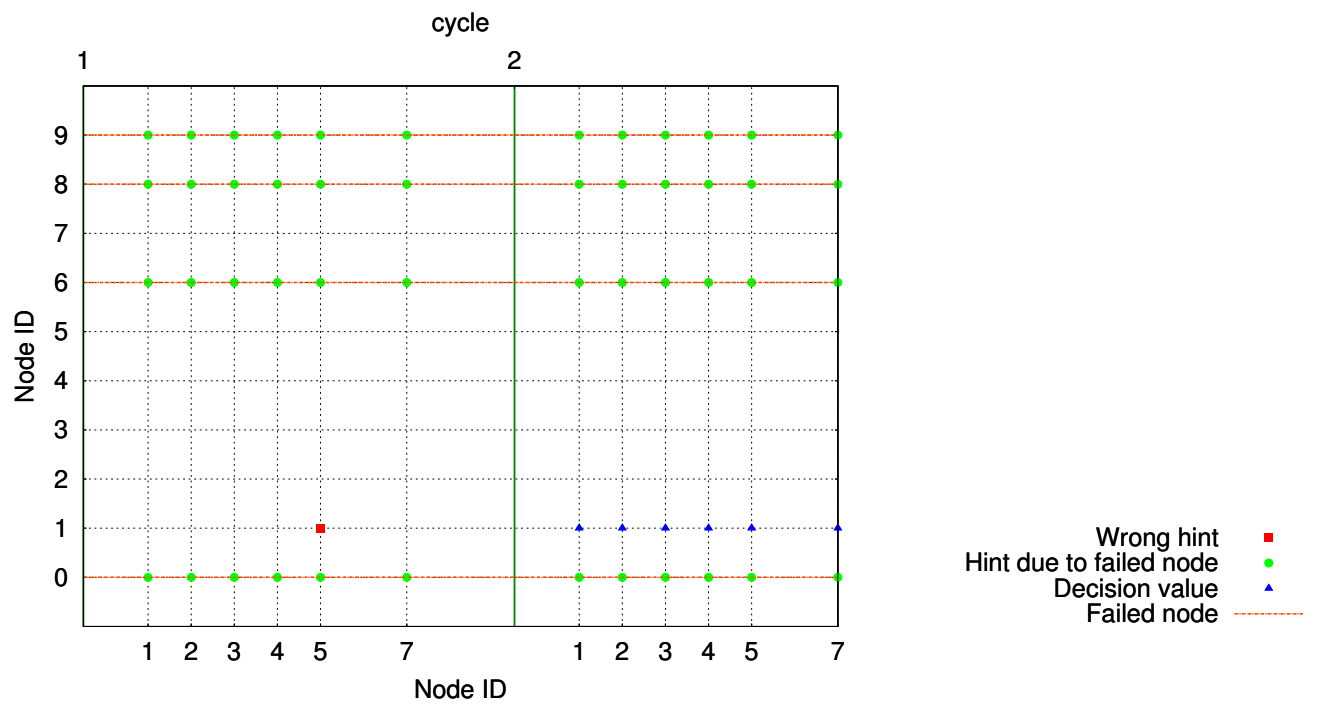


Figure 20: Example of test with 4 failed nodes, summarizing picture about the nodes status

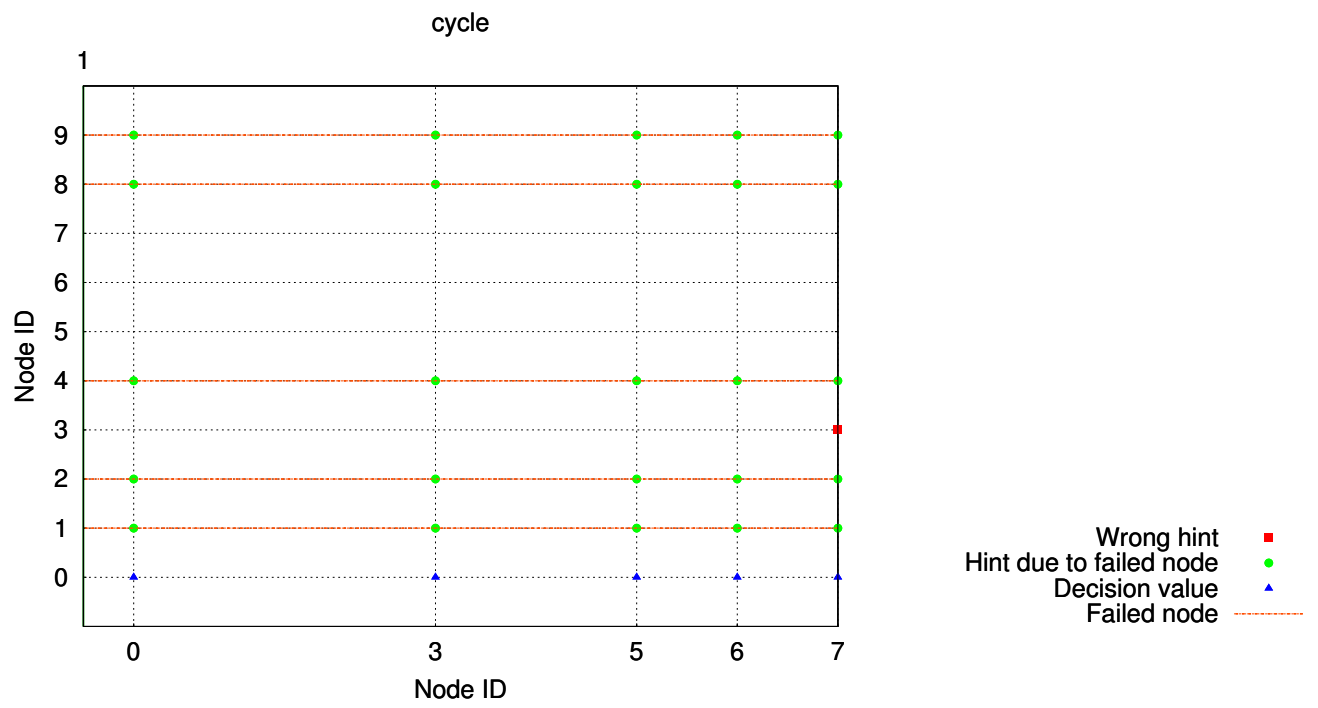


Figure 21: Example of test with 5 failed nodes, summarizing picture about the nodes status

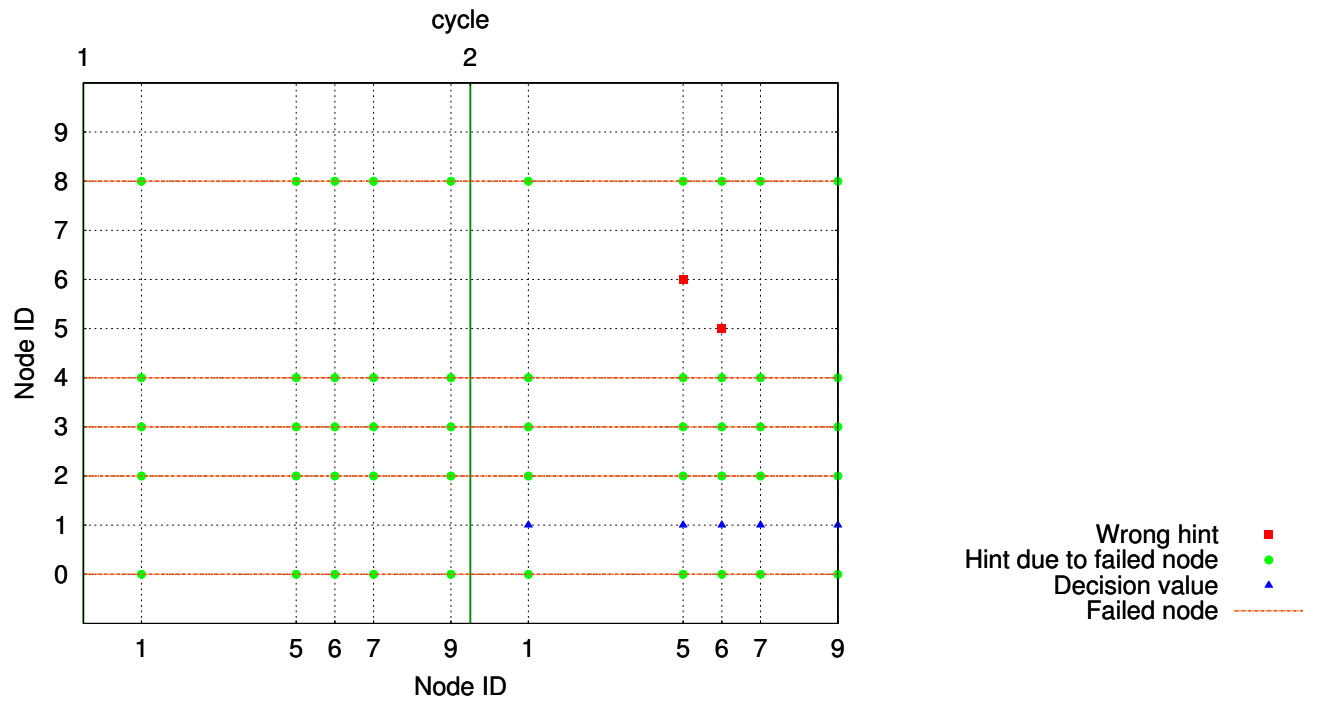


Figure 22: Example of test with 5 failed nodes, summarizing picture about the nodes status

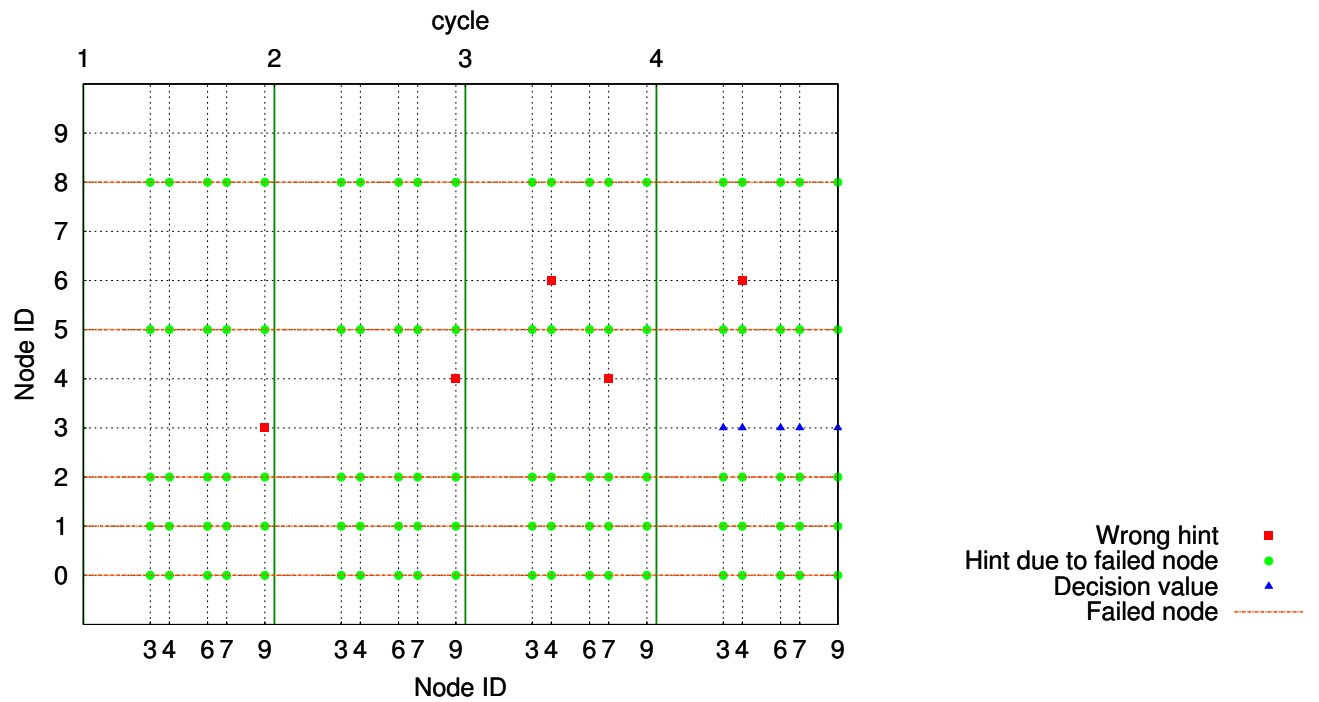


Figure 23: Example of test with 5 failed nodes, summarizing picture about the nodes status

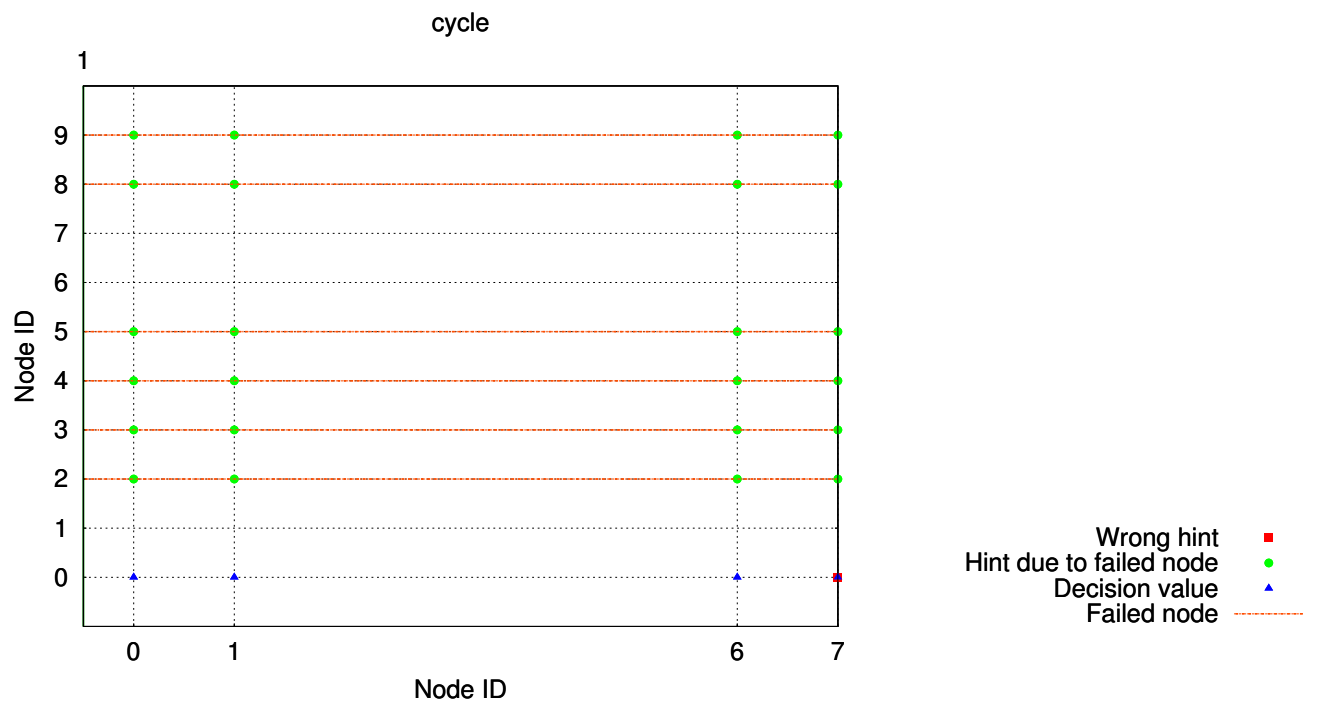


Figure 24: Example of test with 6 failed nodes, summarizing picture about the nodes status

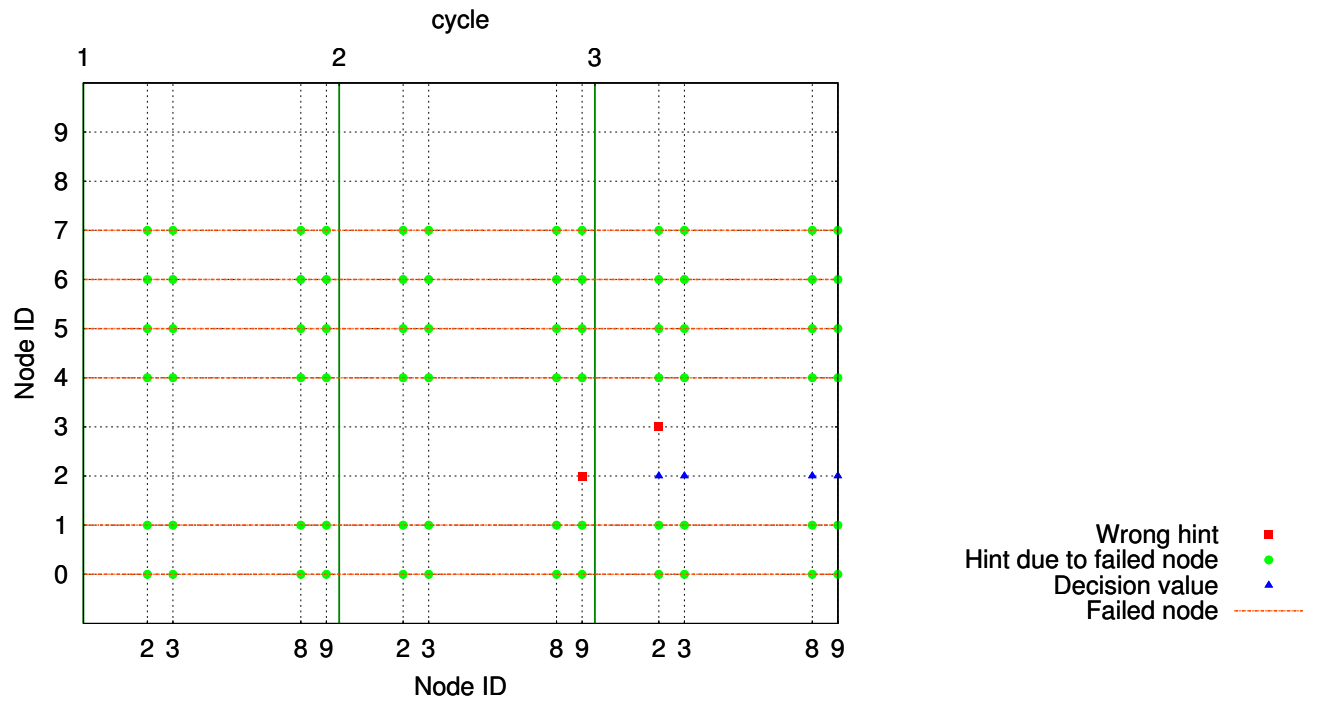


Figure 25: Example of test with 6 failed nodes, summarizing picture about the nodes status

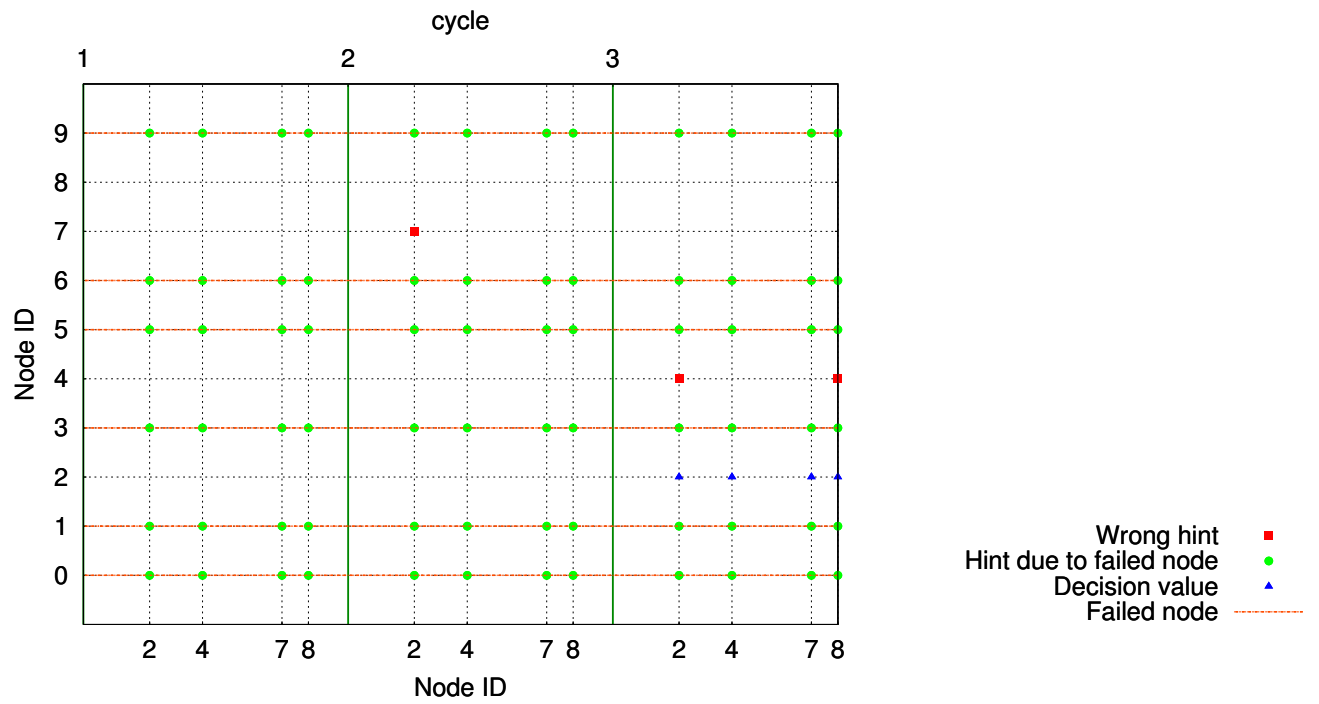


Figure 26: Example of test with 6 failed nodes, summarizing picture about the nodes status

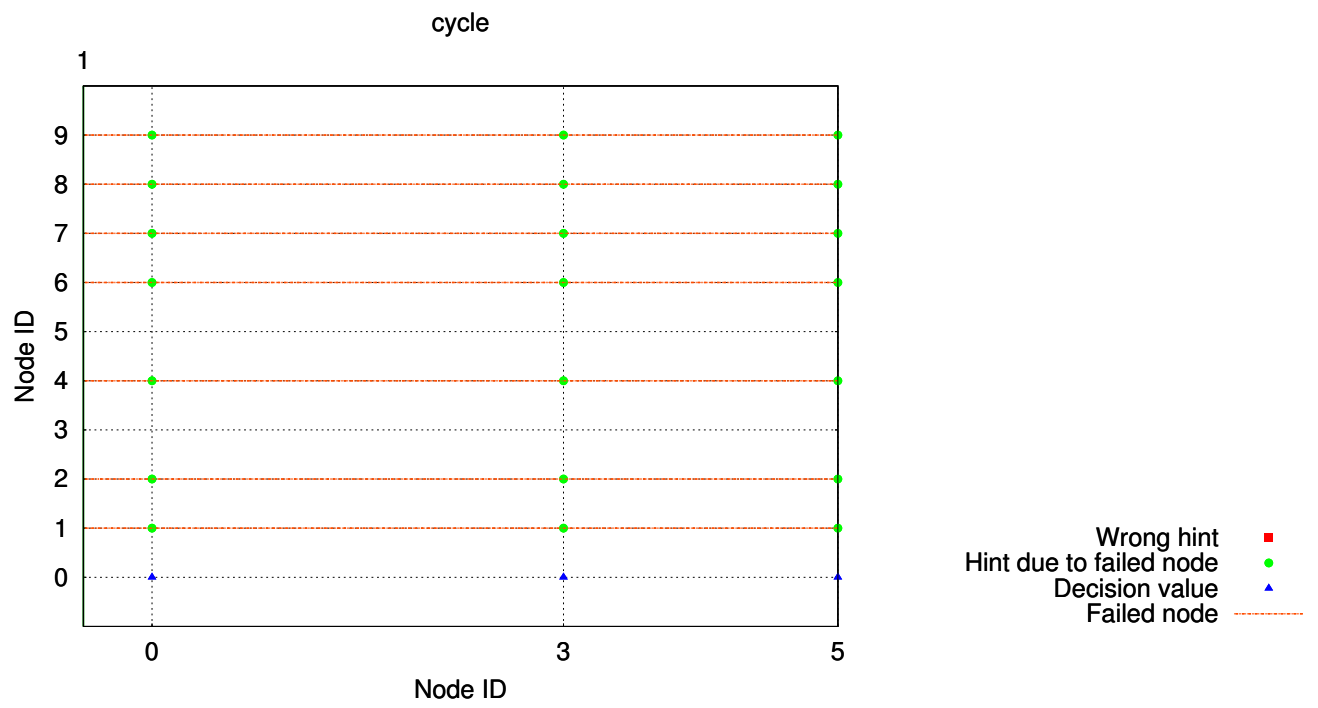


Figure 27: Example of test with 7 failed nodes, summarizing picture about the nodes status

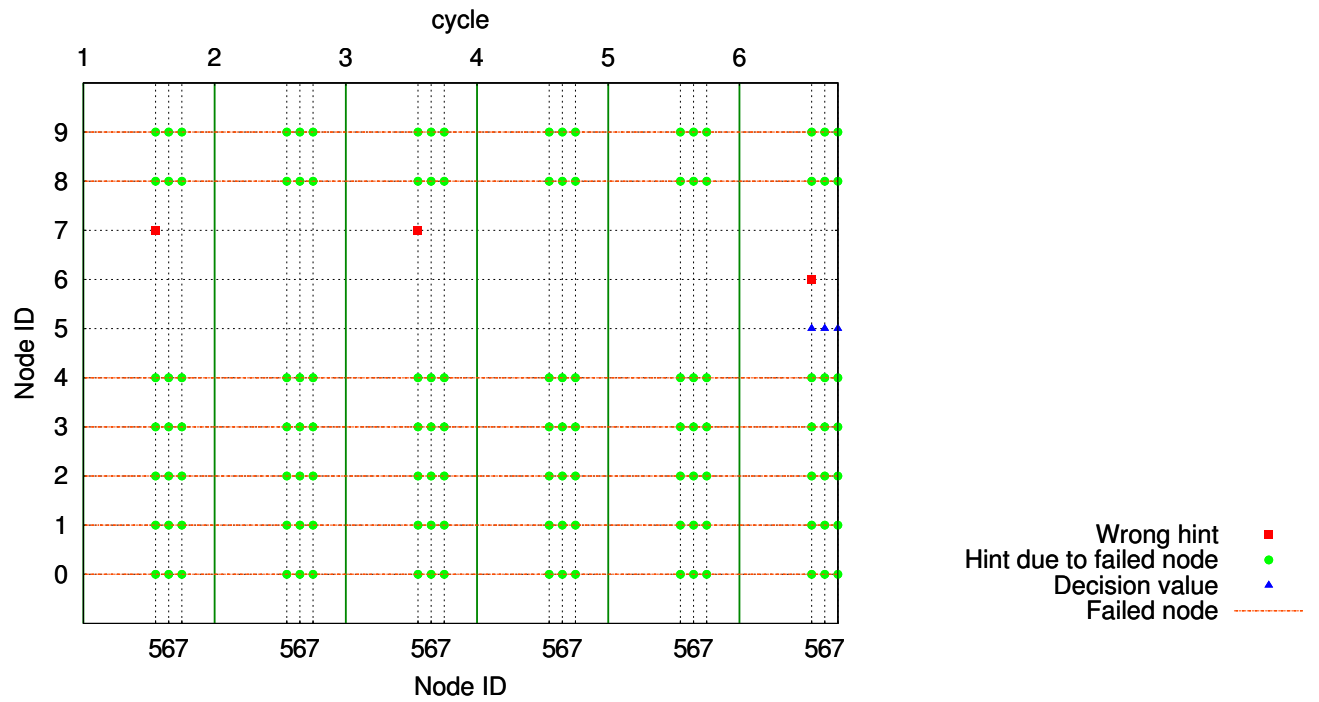


Figure 28: Example of test with 7 failed nodes, summarizing picture about the nodes status

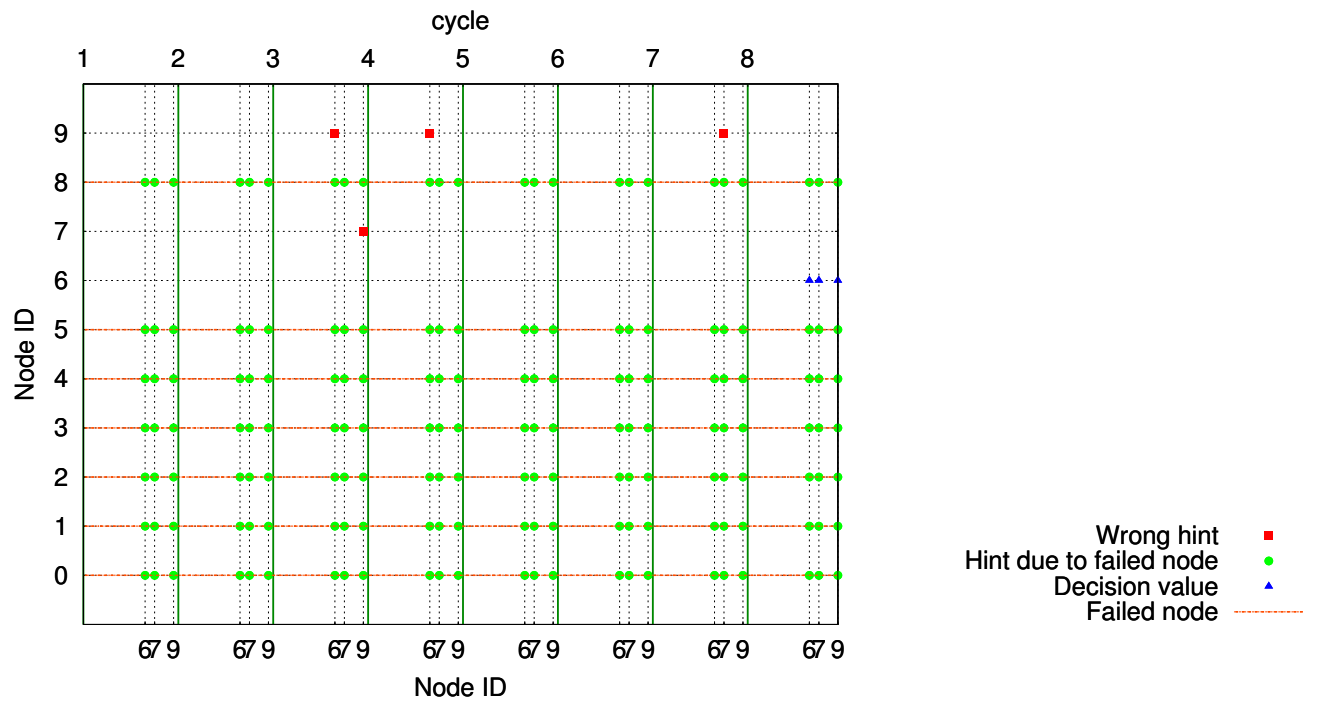


Figure 29: Example of test with 7 failed nodes, summarizing picture about the nodes status

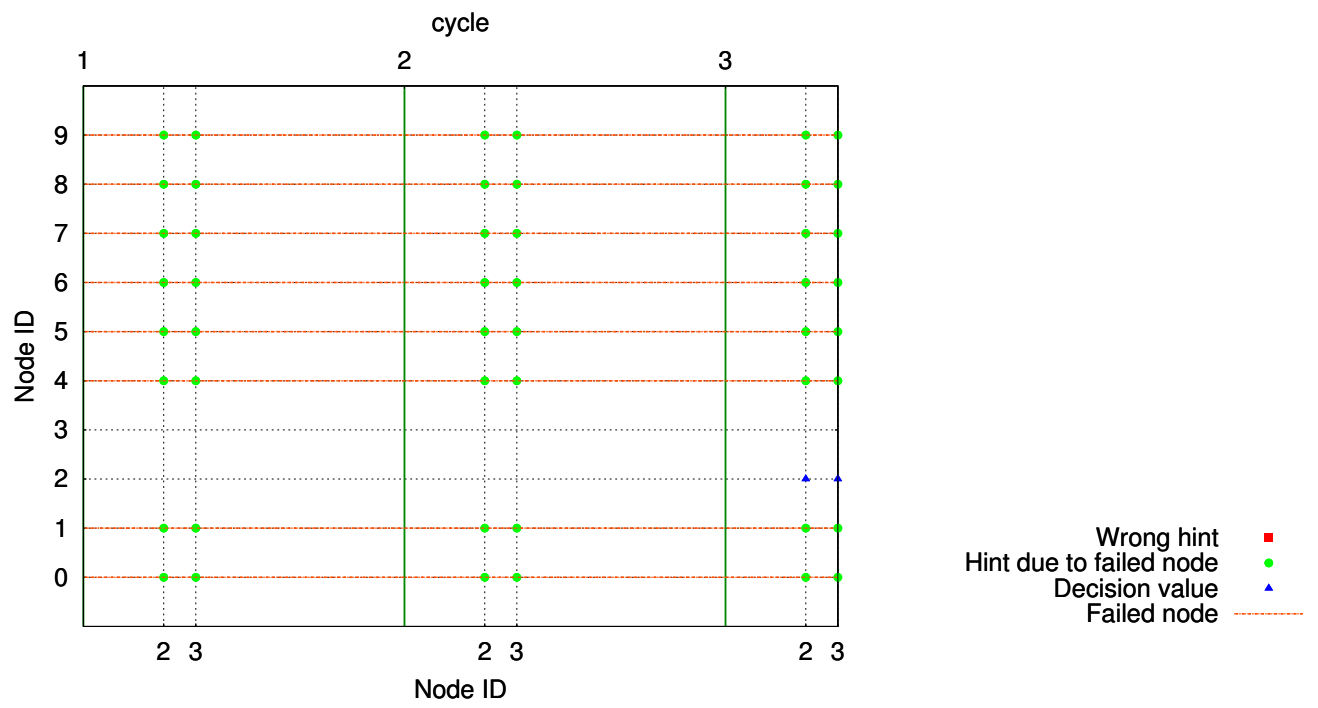


Figure 30: Example of test with 8 failed nodes, summarizing picture about the nodes status

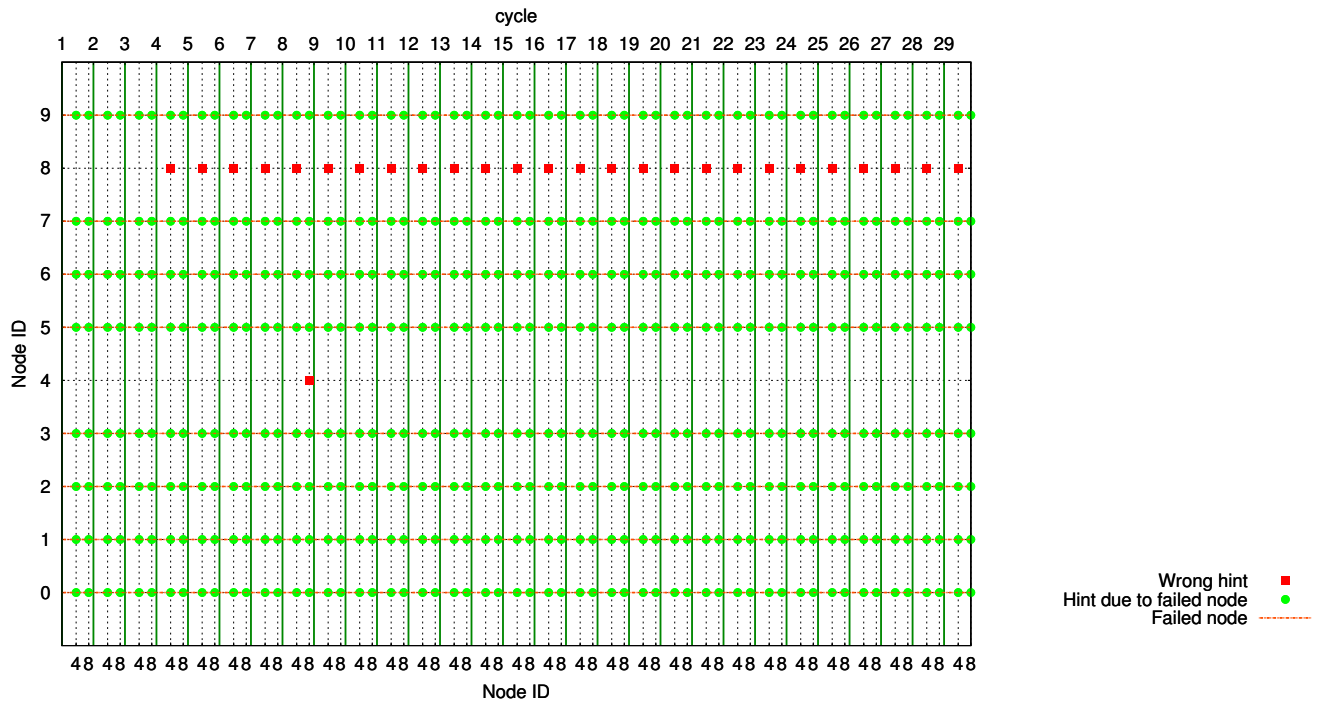


Figure 31: Example of test with 8 failed nodes, summarizing picture about the nodes status

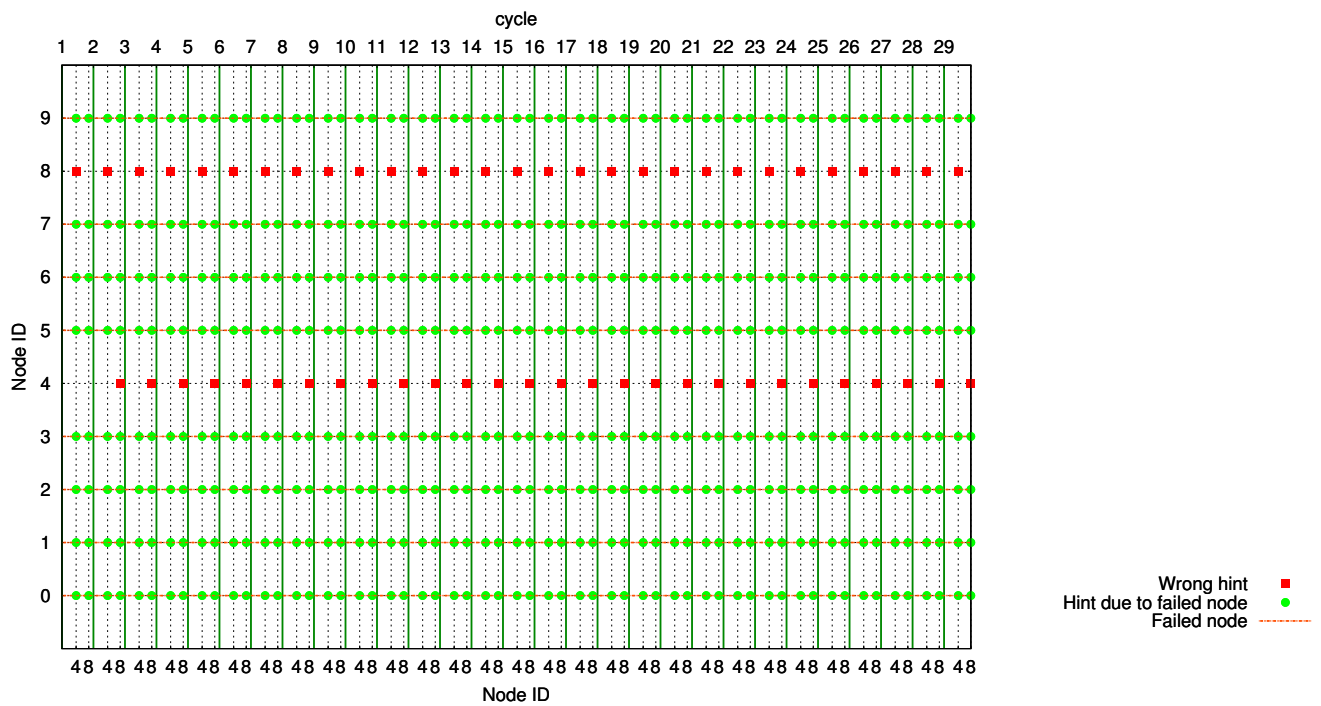


Figure 32: Example of test with 8 failed nodes, summarizing picture about the nodes status

## A.2 Consensus with Eventually Strong FD

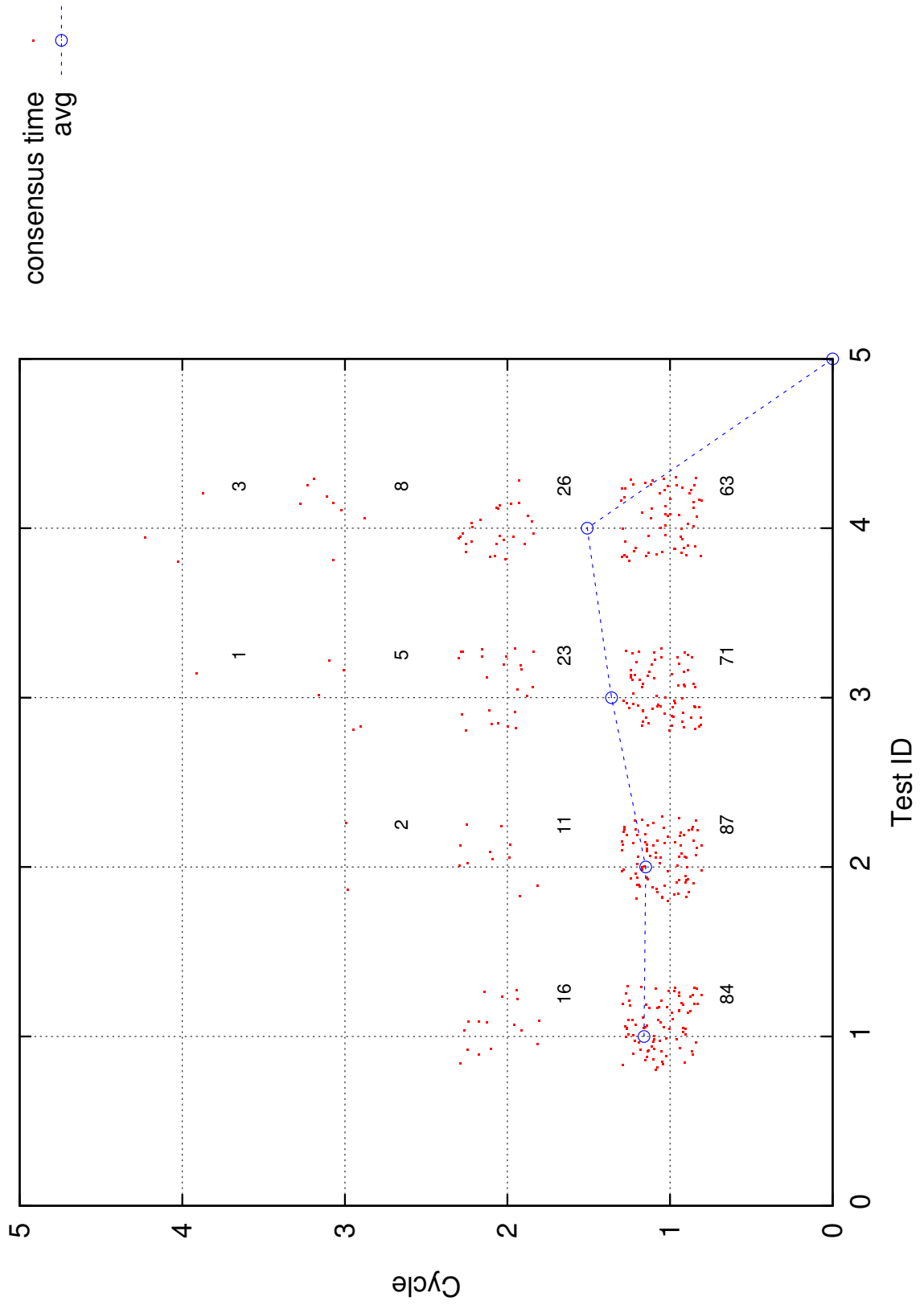


Figure 33: Consensus times

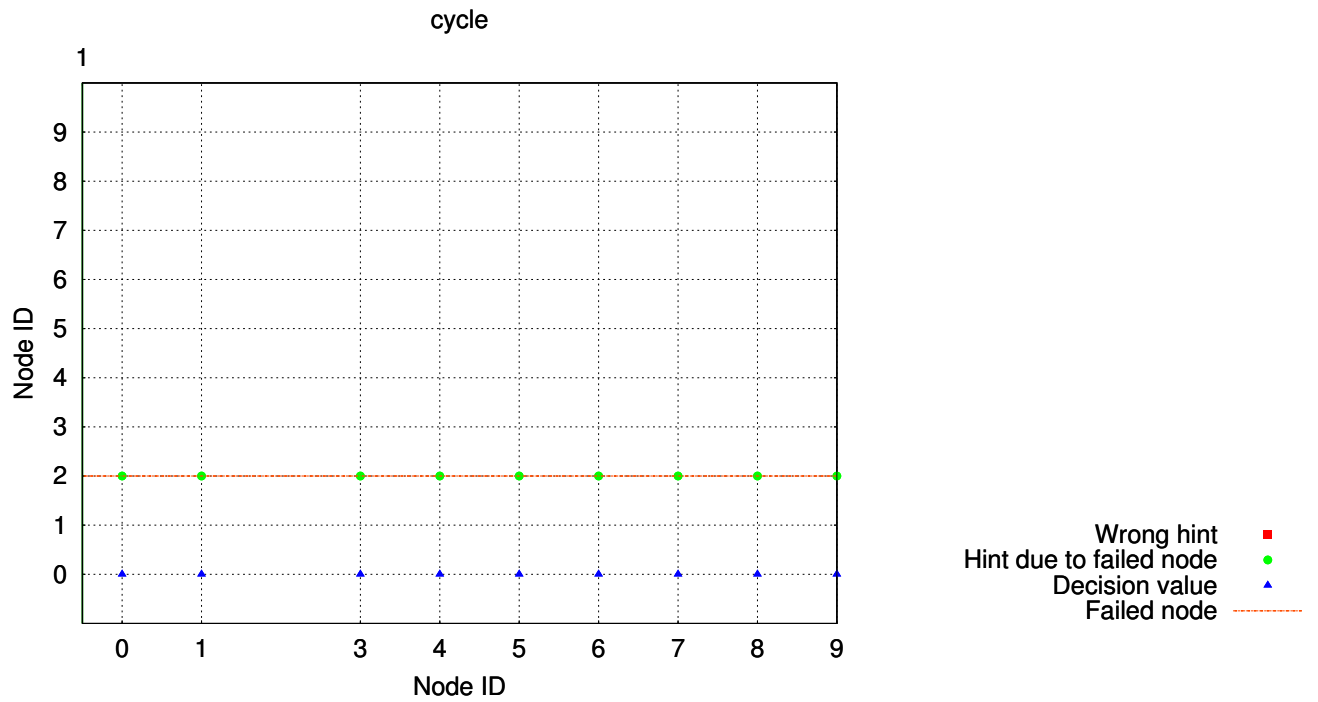


Figure 34: Example of test with 1 failed node, summarizing picture about the nodes status

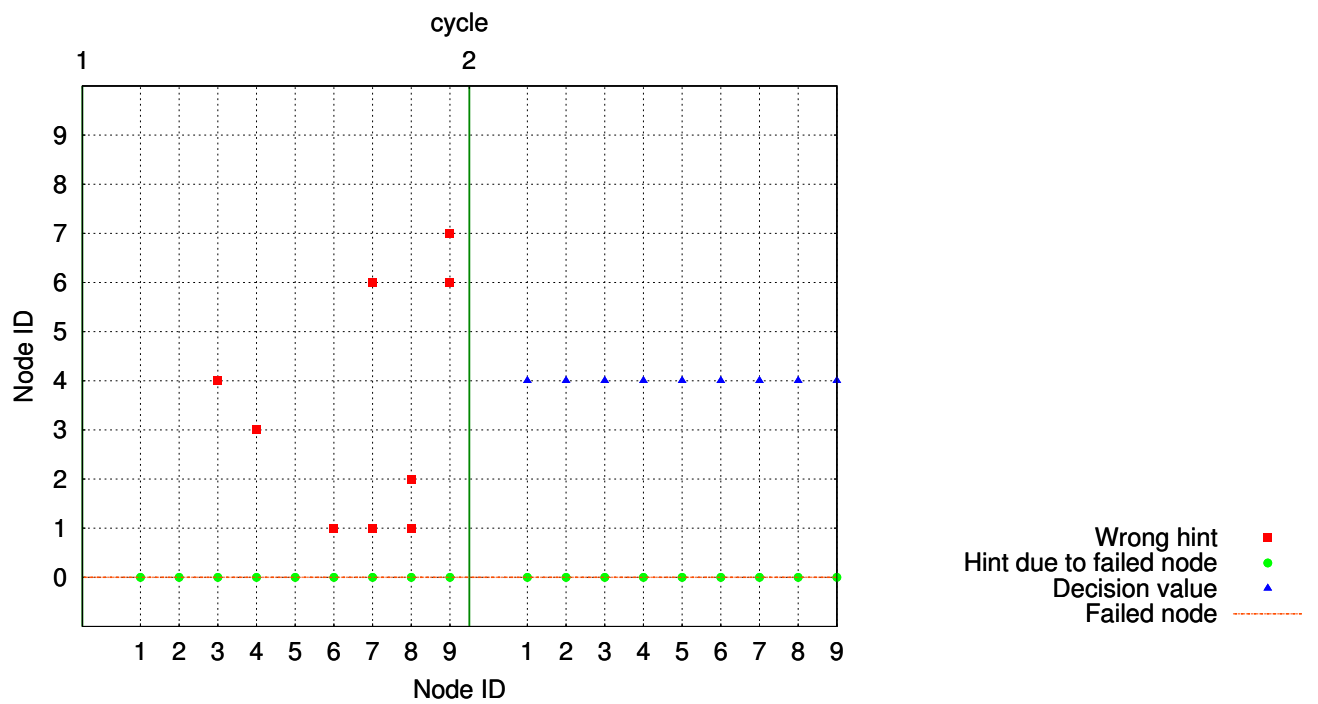


Figure 35: Example of test with 1 failed node, summarizing picture about the nodes status

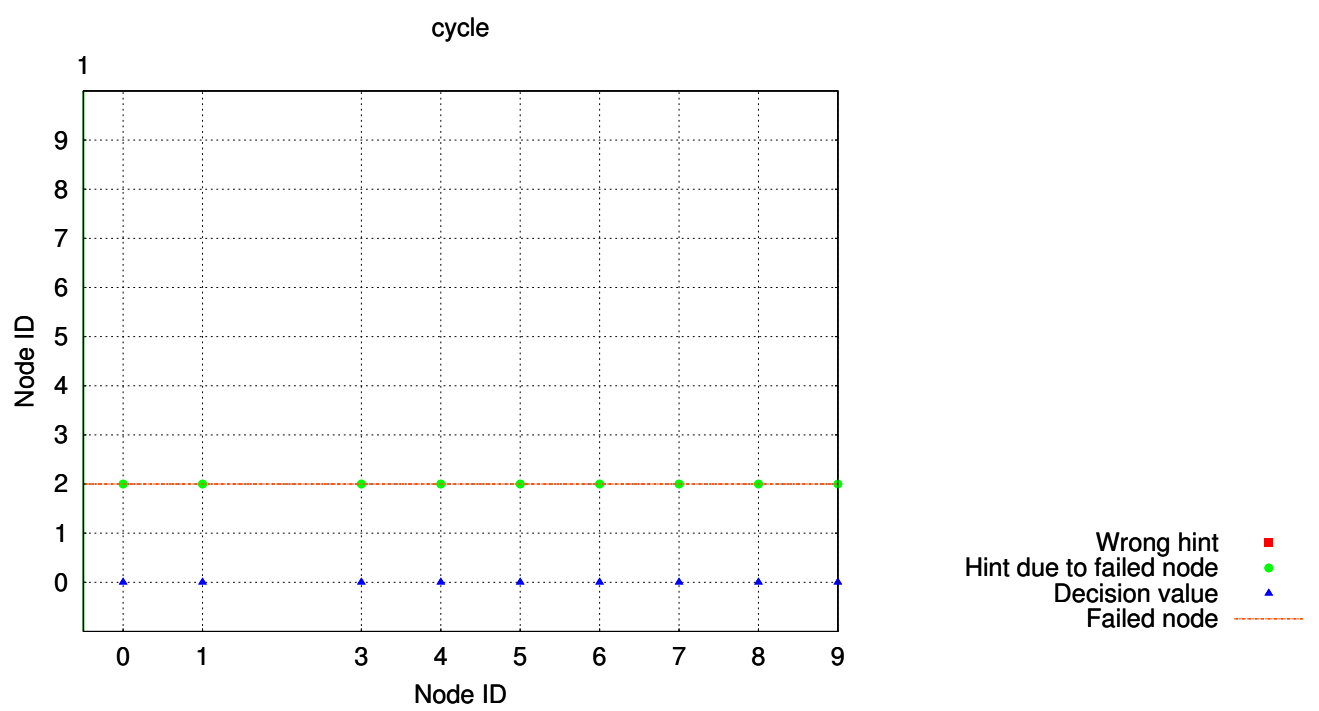


Figure 36: Example of test with 1 failed node, summarizing picture about the nodes status

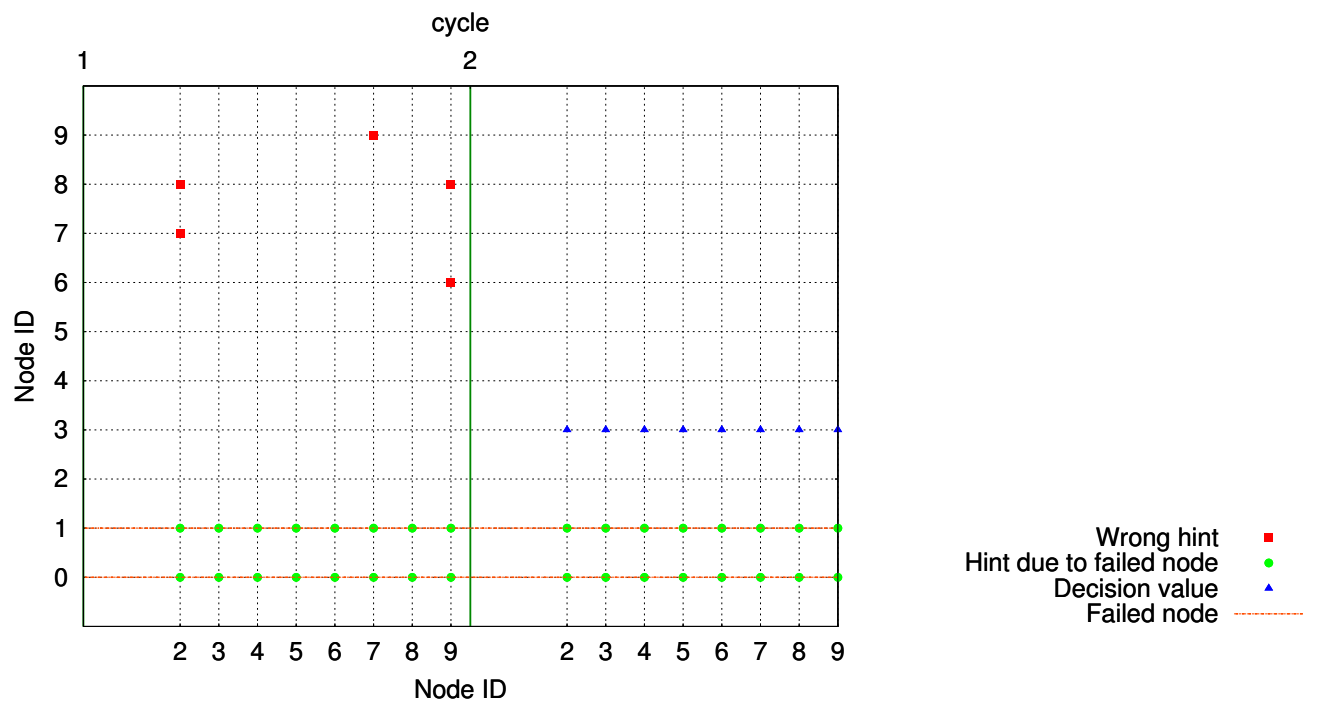


Figure 37: Example of test with 2 failed nodes, summarizing picture about the nodes status

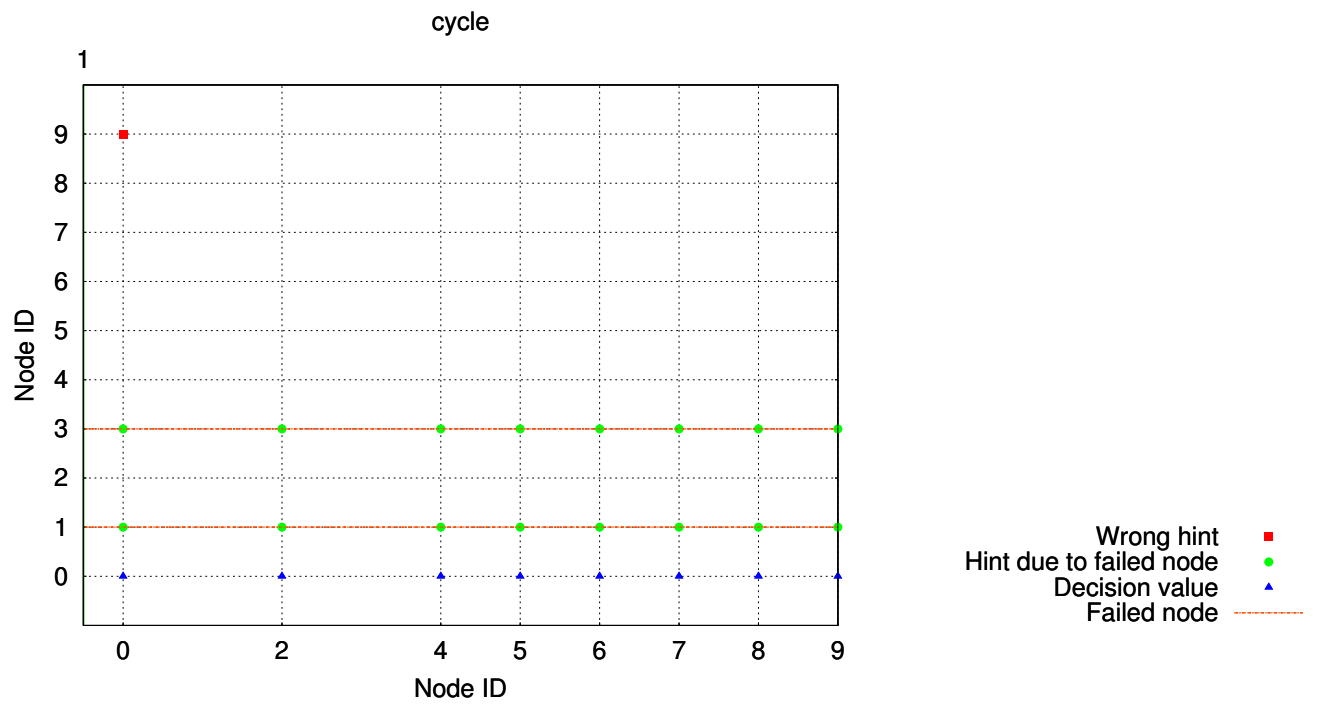


Figure 38: Example of test with 2 failed nodes, summarizing picture about the nodes status

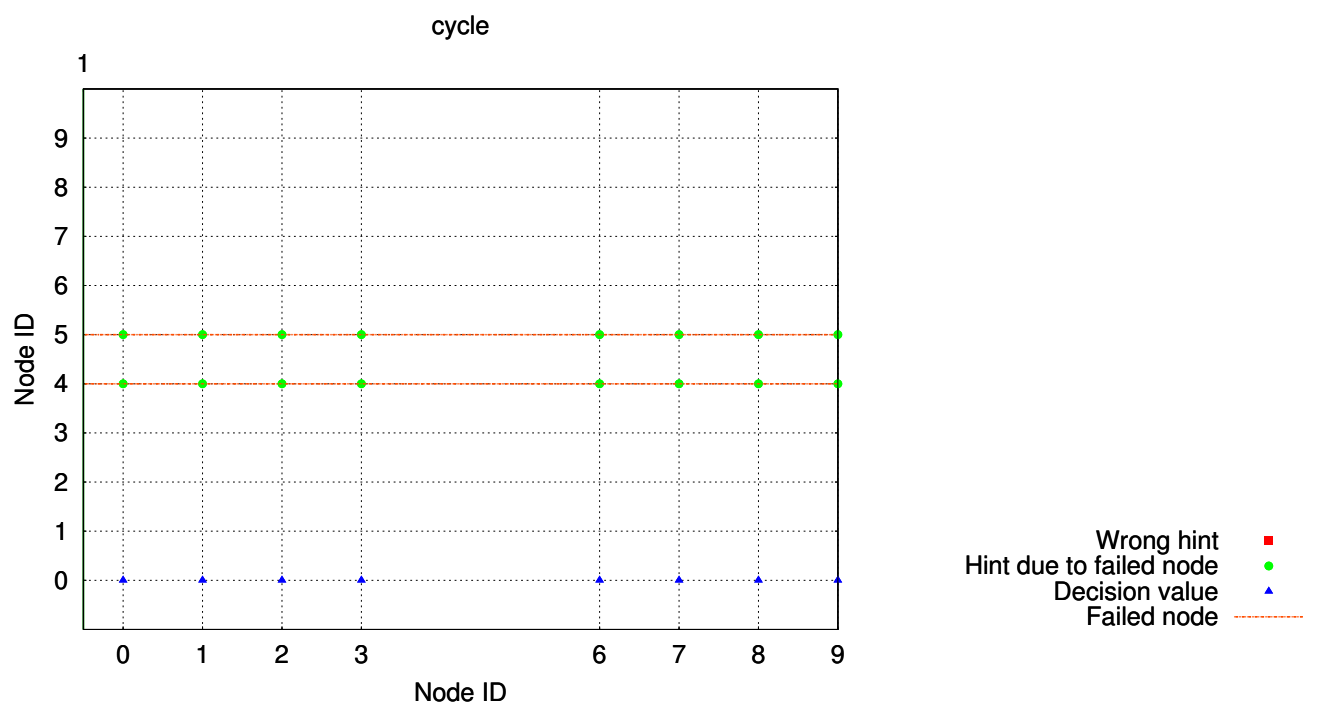


Figure 39: Example of test with 2 failed nodes, summarizing picture about the nodes status

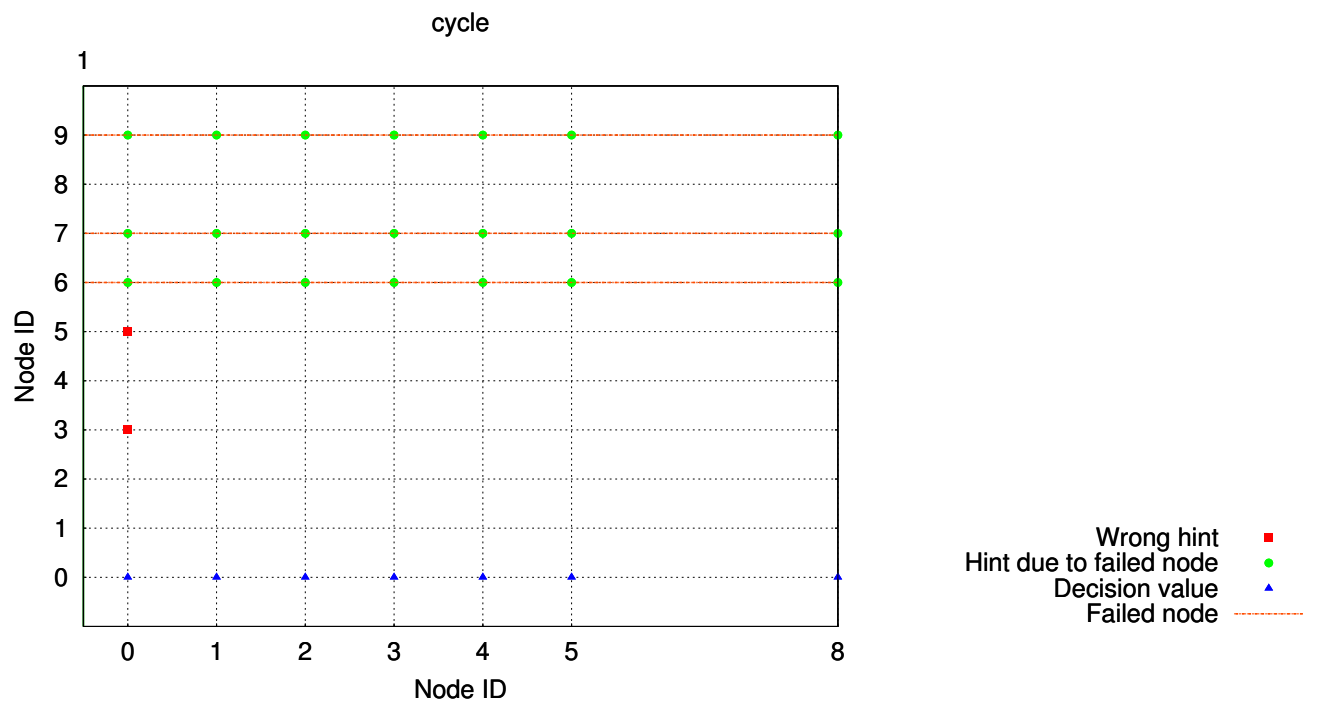


Figure 40: Example of test with 3 failed nodes, summarizing picture about the nodes status

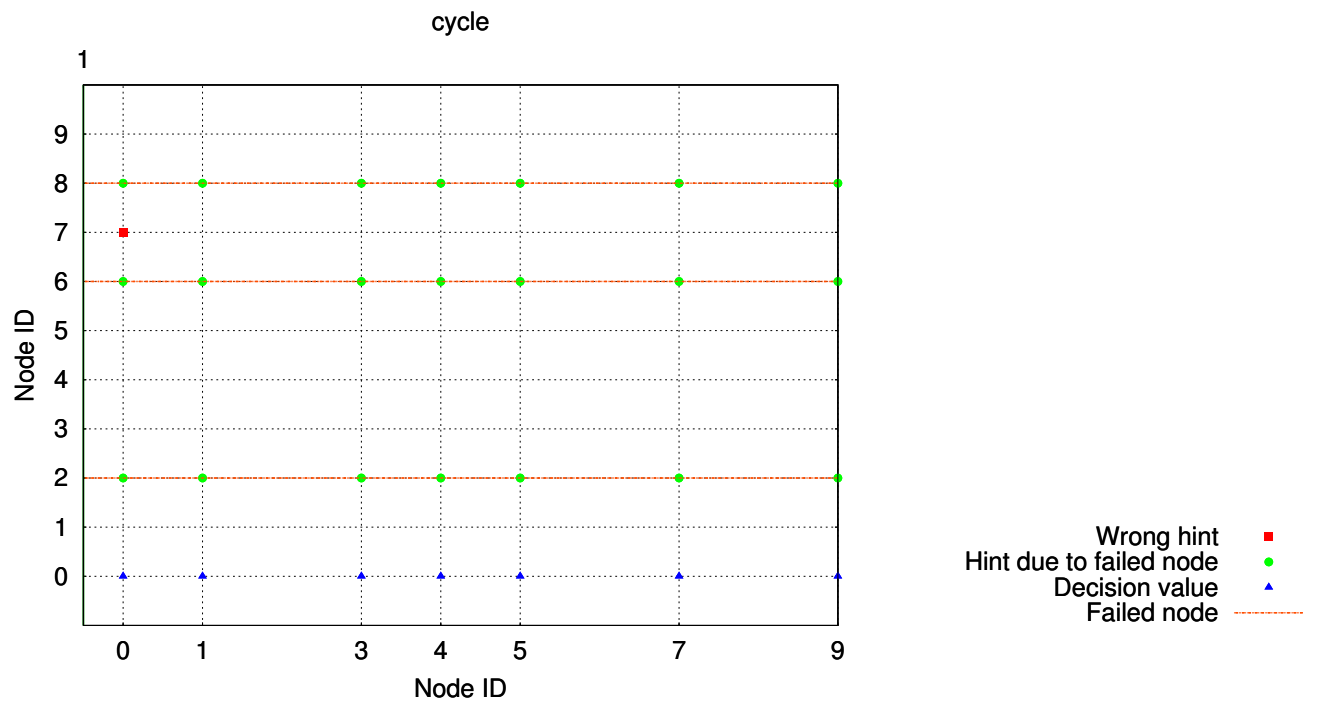


Figure 41: Example of test with 3 failed nodes, summarizing picture about the nodes status

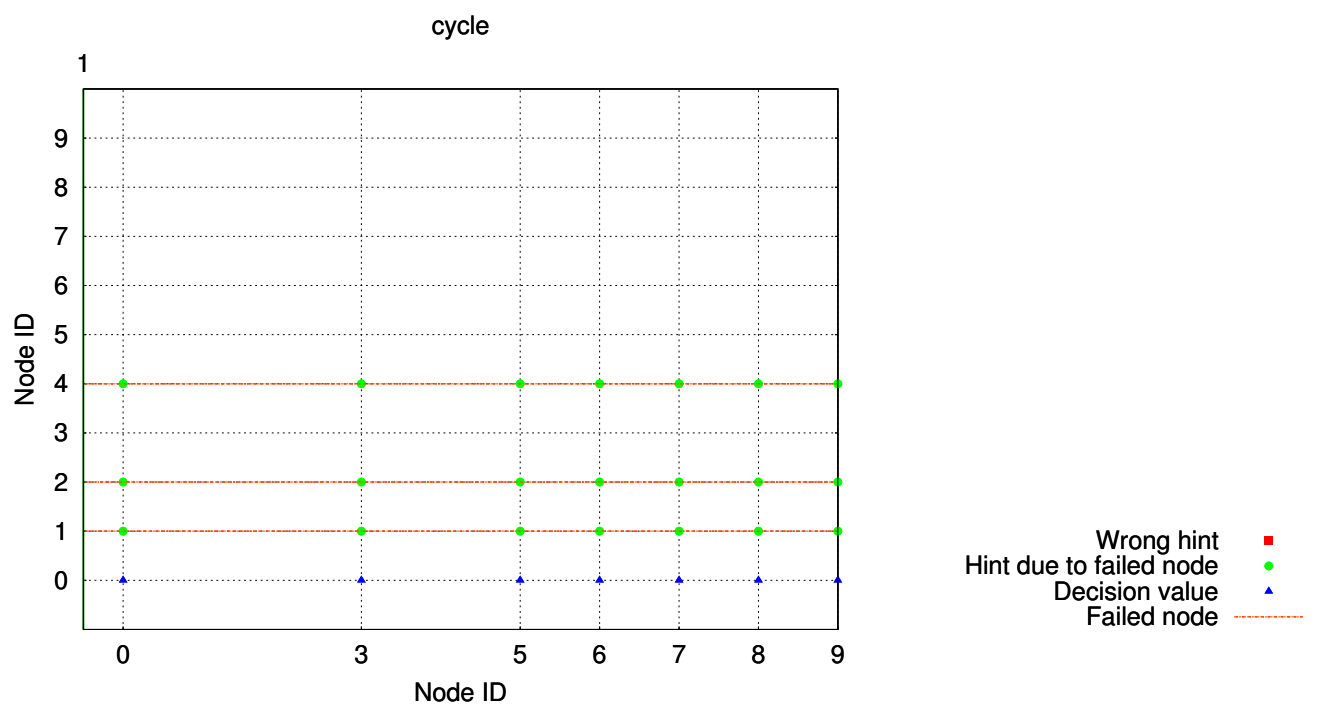


Figure 42: Example of test with 3 failed nodes, summarizing picture about the nodes status

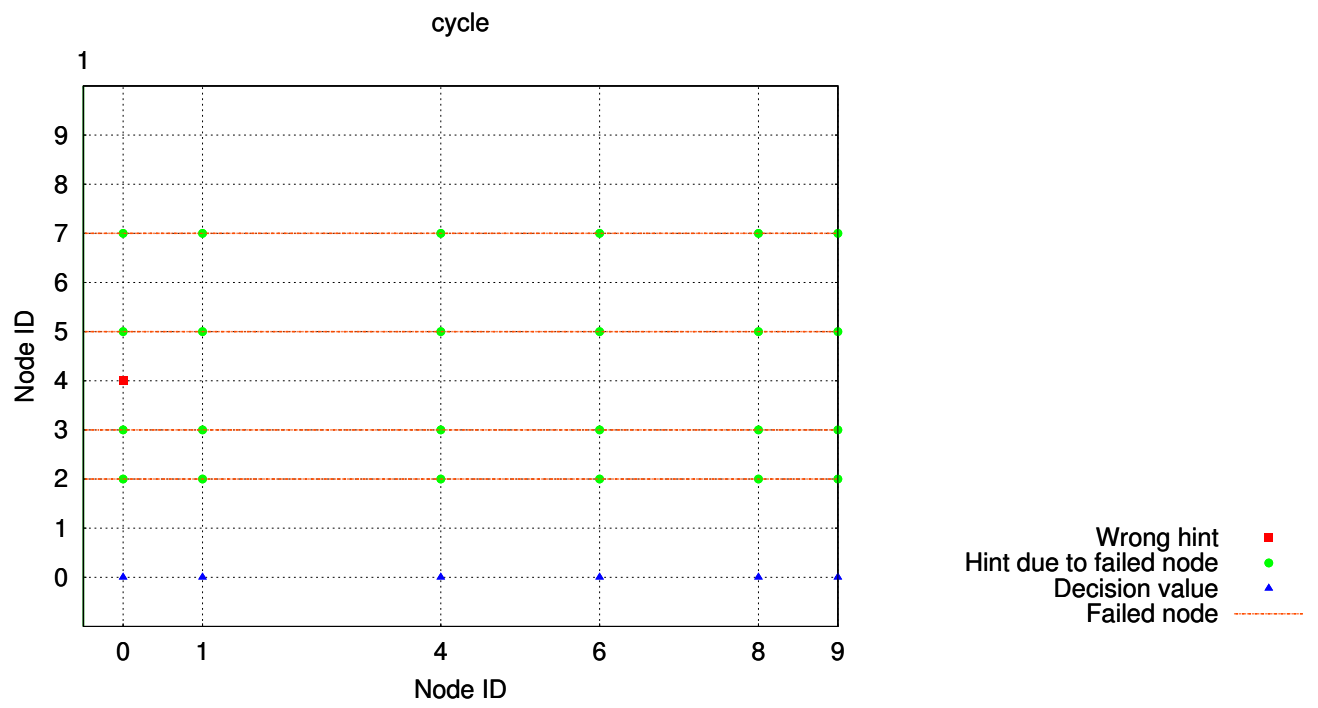


Figure 43: Example of test with 4 failed nodes, summarizing picture about the nodes status

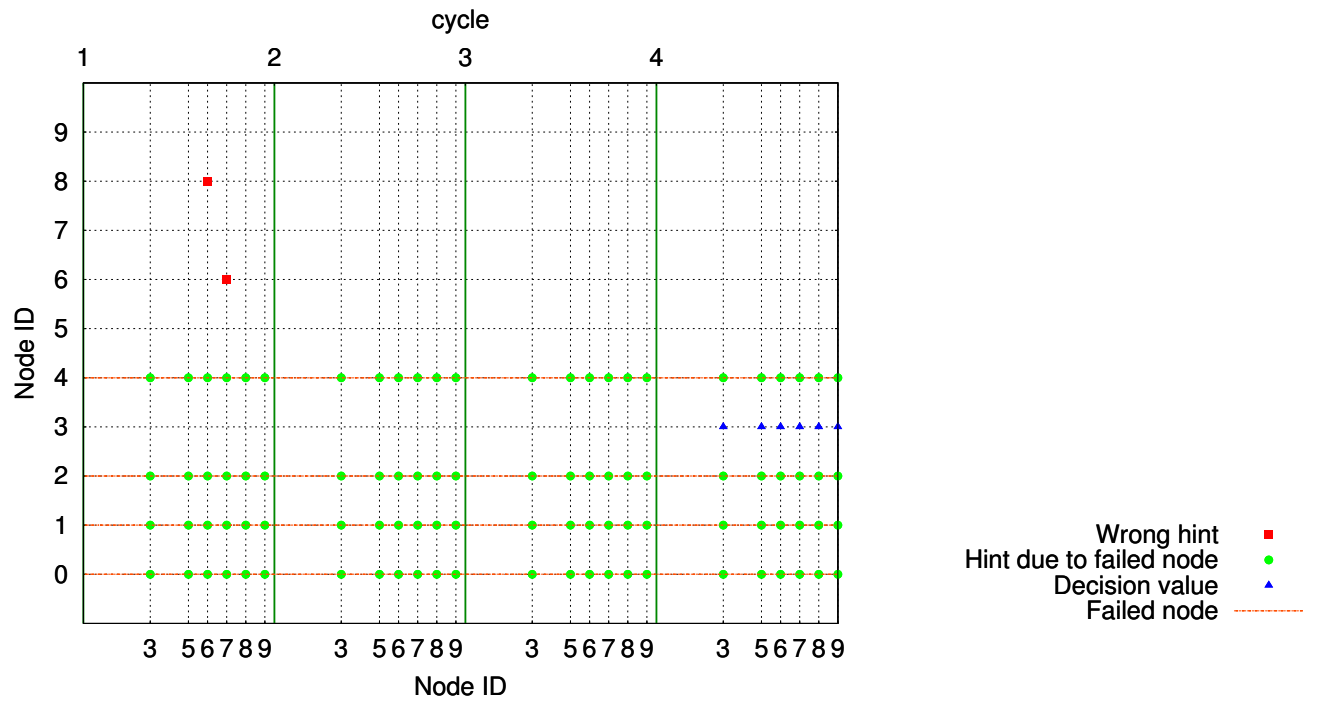


Figure 44: Example of test with 4 failed nodes, summarizing picture about the nodes status

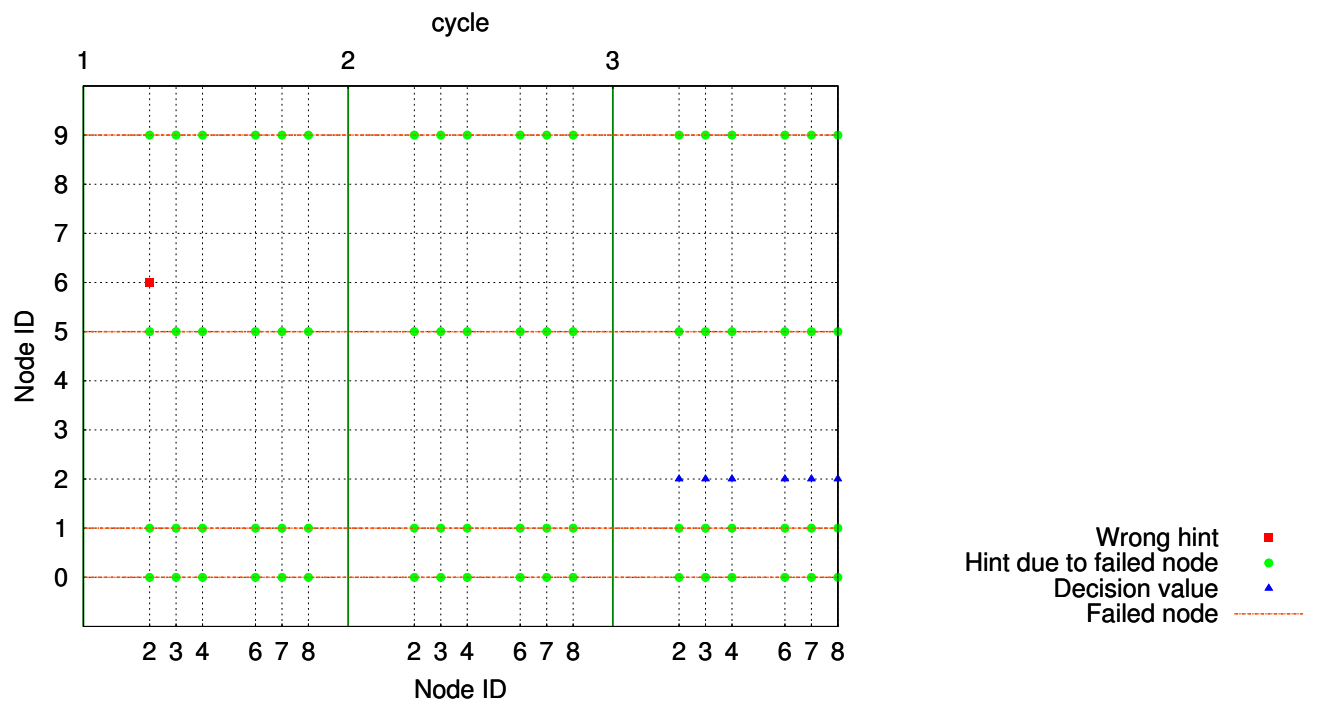


Figure 45: Example of test with 4 failed nodes, summarizing picture about the nodes status

## References

- [1] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
- [2] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, apr 1985.
- [3] Mark Jelasity. *A Basic Event Driven Example for PeerSim 1.0*, 1.0 edition, November 2006. URL <http://peersim.sourceforge.net/tutorialed/tutorialed.pdf>. [Online; accessed 14-July-2010].
- [4] Gian Paolo Jesi. *PeerSim HOWTO: Build a new protocol for the PeerSim 1.0 simulator*. University of Bologna, December 2005. URL <http://peersim.sourceforge.net/tutorial1/tutorial1.pdf>. [Online; accessed 14-July-2010].
- [5] Alberto Montresor and Mark Jelasity. Peersim: A scalable p2p simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, sep 2009.
- [6] Achour Mostéfaoui and Michel Raynal. Solving consensus using chandra-toueg’s unreliable failure detectors: A general quorum-based approach. In *In Proceedings of the 13th International Symposium on Distributed Computing (DISC'00)*, pages 49–63, Bratislava, Slovak Republic, 1999. URL [disi.unitn.it/~montreso/ds/syllabus/papers/PI-1254.pdf](http://disi.unitn.it/~montreso/ds/syllabus/papers/PI-1254.pdf).
- [7] Colin Kelley Thomas Williams. Gnuplot. URL <http://www.gnuplot.info/>. [Online; accessed 10-May-2011].