Fault-tolerant Distributed Continuous Double Auctioning on Computationally Constrained Microgrids

Anesu M. C. Marufu[1], Anne V. D. M. Kayem[1] and Stephen Wolthusen[2,3]
[1]*Department of Computer Science, University of Cape Town, Rondebosch 7701, Cape Town, South Africa*
[2]*School of Mathematics and Information Security, Royal Holloway, University of London, London, U.K.*
[3]*Norwegian Information Security Laboratory, Department of Computer Science, Gjovik University College, Gjovik, Norway*

Keywords: Computationally Constrained Microgrid, Power Network Stability, Fault Tolerance, Auction Protocol.

Abstract: In this article we show that a mutual exclusion protocol supporting continuous double auctioning for power trading on computationally constrained microgrid can be fault tolerant. Fault tolerance allows the CDA algorithm to operate reliably and contributes to overall grid stability and robustness. Contrary to fault tolerance approaches proposed in the literature which bypass faulty nodes through a network reconfiguration process, our approach masks crash failures of cluster head nodes through redundancy. Masking failure of the main node ensures the dependent cluster nodes hosting trading agents are not isolated from auctioning. A redundant component acts as a backup which takes over if the primary components fails, allowing for some fault tolerance and a graceful degradation of the network. Our proposed fault-tolerant CDA algorithm has a complexity of $O(N)$ time and a check-pointing message complexity of $O(W)$. N is the number of messages exchanged per critical section. W is the number of check-pointing messages.

1 INTRODUCTION

Auction mechanisms for resource allocation support power trading schemes on microgrids (Cui et al., 2014), (Borenstein et al., 2002), (Izakian et al., 2010), (Pałka et al., 2012), (Stańczak et al., 2015) (Marufu et al., 2015). Most microgrid power trading schemes assume network reliability. When computationally constrained devices form the infrastructure of such networks, problems ranging from signal distortion to component failures emerge. (Marufu et al., 2015) proposed a distributed Continuous Double Auction (CDA) algorithm for allocating power in a computationally resource-constrained microgrid. The CDA algorithm describes a market scenario in which trading agents sell goods (sellers) and buy goods (buyers). The distributed CDA algorithms ensures efficient message passing(Marufu et al., 2015) and minimal trade execution time. However, the (Marufu et al., 2015) CDA algorithm is not fault tolerant. Fault tolerance reinforces grid resilience.

Fault tolerant systems are able to provide services in the presence of failure (Jalote, 1994) (Médard and Lumetta, 2003). Node failure is performance detrimental to the CDA mechanism. Furthermore, unhandled node failures open avenues for attacks, such as denial of service attacks (DoS) that take advantage of these scenarios. Another supporting notion of fault tolerance within CDA specifications for microgrids stems from Murphy's Law (Bloch, 2003). If we consider the recent precedence for adaptations of CDA mechanisms in solving new problems within similar cyber-physical critical systems (Marufu et al., 2015), the consequences of failure can be serious if fault prevention and tolerance measures are not implemented.

Specifically, we consider a scenario where a head cluster node within a hierarchically structured network architecture fails. Such a failure implies that the child cluster nodes will be isolated from the rest of the network. In addition, failure of the head cluster node may also result in partitioning of the network. If a mutually exclusive distributed CDA mechanism is employed, the isolated nodes which act as hosts for the Trading Agents (*TA*s) will inevitably be excluded from trading. For instance, if pivotal seller *TA*s fail to participate in the auction market, a significant shift of demand over supply occurs which will inflate the energy prices for buyers. This disrupts grid stability (since failure of *TA*s to participate in the auction results uneven balancing of demand and supply) and in-turn causes distrust (due to unreliable energy allocation) among members within the microgrid.

Marufu, A., Kayem, A. and Wolthusen, S.
Fault-tolerant Distributed Continuous Double Auctioning on Computationally Constrained Microgrids.
DOI: 10.5220/0005744304480456
In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 448-456
ISBN: 978-989-758-167-0

In this paper we propose a fault tolerant distributed CDA algorithm that is efficient in message exchange and computation time. In contrast to the standard approach of bypassing faulty nodes, our approach masks failure through redundancy of cluster head nodes.

The remainder of this paper is organised as follows: Section 2 discusses the state of the art. Section 3 presents an overview of the distributed CDA framework for a resource-constrained microgrid, while Section 4 describes the fault model, failure handling strategy the fault tolerant algorithm. We analyse the fault tolerant CDA algorithm through correctness and efficiency. Section 5 concludes this article and identifies on-going and future work.

2 RELATED WORK

In (Marufu et al., 2015) a token based mutual exclusion protocol is used to support a CDA auction mechanism, which means the suggested CDA formulation inherits properties of the distributive primitives. Requests to enter the CS are directed to whichever node is the current token holder. The token-based mutual exclusion approach adopted in (Marufu et al., 2015) was inspired by (Raymond, 1989). Raymond's algorithm is resilient to non-adjacent node crashes and recoveries, but not node/link failures. As an extension to Raymond's algorithm, (Chang et al., 1990) imposed a logical direction on a number of edges to induce a token oriented *directed acyclic graph* (DAG), where there exists a directed path originating from n while terminating at the token holding node. Resilience to link and site failures is achieved by allowing request messages to be sent over all edges of the DAG. Besides high message complexity costs, the solution in (Chang et al., 1990) does not consider link recovery. In related work (Dhamdhere and Kulkarni, 1994) reveal that the algorithm in (Chang et al., 1990) can suffer from deadlock, thereby they propose a dynamically changing sequence number to each node to form a total ordering of the nodes. Since the token holding node possesses the highest sequence number, a DAG is maintained if the links are defined to point to a node with higher sequence number. In cases where a node has no outgoing links to the token holder (due to link failure), it will flood the network with messages to build a spanning tree. Once the token is part of this spanning tree, a token will be granted to this node, bypassing other requests. If link failures are persistent (as will be the case in our given context) starvation will be inevitable since priority is given to nodes that lose a path to the token holding node. In addition, flooding of messages incurs high costs for a typical resource constrained setup. (Walter et al., 2001) present a reversal of the technique where a destination oriented DAG is maintained when destination is a dynamic destination. Ordering of nodes is similar as in (Dhamdhere and Kulkarni, 1994), but the lowest node is assigned the token. The described works mainly address the issue of link failure with little effort on addressing node failure, explicitly assuming that nodes do not fail and network partitioning will not occur. Another body of literature indicates that some work towards solving node failure has been done. (Revannaswamy and Bhatt, 1997) propose a solution to handle failures of nodes and links in a network with definitions carried over from (Raymond, 1989). Their attempt at fault tolerance involves eliminating the failed nodes and obtaining a different tree structure utilising chords (edges of the tree yet unused for message exchanges) from the network. A reconfiguration process attempts to connect parts of the tree separated due to failures. Reconfiguration eliminates failed components, when the cluster node is affected, the nodes that are reliant on it will not be able to participate in the bidding process. This is undesirable and so a robust solution that is able to guarantee reliability is needed.

3 COMPUTATIONALLY CONSTRAINED CONTEXT

We consider a small community microgrid supported by a double auctioning layer. An energy sharing agreement amongst participating community members ensures fair and trustworthy access to energy. The community is comprised of a number of clustered households within a given area. Distribution of energy is facilitated by a communication network that overlays a power distribution network. This communication network is hierarchically structured with a large number of mobile phones M_{mp} forming cluster nodes and relatively fewer, fixed smart meters M_{sm} forming cluster heads. We consider that M_{mp} nodes host the trading agents TA_n which conduct trade in the CDA market on behalf of the community members.

To ensure that only one trading agent submits an offer in the auction market at a time, (Marufu et al., 2015) propose a protocol for the serialization of market access and fair chances to submit an offer. The protocol satisfies the following properties:

- ME1: [Mutual exclusion] At most only one trading agent can remain in the auction market at any time. This is a safety property.

- ME2: [Freedom from deadlock] When the auction market is free, one of the requesting trading agents should enter critical section. This is a liveness property.
- ME3: [Freedom from Starvation] A trading agent should not be forced to wait indefinitely to enter the auction market while other trading agents are repeatedly executing their requests. This is a liveness property.
- ME4: [Fairness] requests must be executed in the order in which they are made.

The algorithm in (Marufu et al., 2015) uses a token-based approach to address the mutual exclusion problem because of the low message traffic generally associated with such algorithms (Ghosh, 2014) (Raymond, 1989) (Raynal, 1986) (Kshemkalyani and Singhal, 2008) (Garg, 2011). A token is a 'permit' passed around among requesting trading agents thereby granting privilege to enter the auction market. Thus, a trading agent requests for the token in order to participate in the auction. At any moment, only one trading agent exclusively holds the token. The token contains a copy of the order-book to which the *TA* submits its offer (Marufu et al., 2015).

Studies indicate that resiliency to failure as one weakness in the adopted token-based approach (Chang et al., 1990) (Dhamdhere and Kulkarni, 1994) (Walter et al., 2001). Although Marufu *et al*'s algorithm (Marufu et al., 2015) includes some fault prevention and to a lesser extent fault tolerance as part of the token-handling specification the Marufu *et al* solution assumes that M_{sm} nodes do not fail. Token-handling in Marufu *et al* includes:

1. Temporarily relaying the token through a close-by, same-cluster $M_{mp}i$ node with better connectivity to M_{sm}.
2. Prompting an $M_{mp}i$ node requesting the token to disable its sleep mode functionality until it has entered the critical section.
3. Employing timing checks to ensure $M_{mp}i$ nodes do not hold on to the token indefinitely as a result of faults or disconnections.

The Marufu *et al*'s CDA fails to guarantee trading agents' participation in the auction market in the event of M_{sm} node failure. The following section discusses the approach we use in order to design a distributed CDA algorithm with fault tolerance.

4 FAULT TOLERANCE

We propose handling crash failures of M_{sm} nodes in the network specifications from (Marufu et al., 2015). As an initial step to building a fault-tolerant system, the literature (Médard and Lumetta, 2003) (Jalote, 1994) (van Steen and Tanenbaum, 2001) (Tanenbaum and Van Steen, 2007) suggests defining the fault model (the number and classes of faults that need to be tolerated). A fault model includes a set of failure scenarios along with frequency, duration and impact of each scenario (Médard and Lumetta, 2003). Inclusion of fault scenarios in the fault model is based on the frequency, impact on the system and feasibility or the cost of providing protection. The next section specifies the fault classes and scenarios we use to formulate the fault tolerant CDA.

4.1 Fault Model

Among the several well-known failure classification schemes, this article considers crash failures, omission failures, timing failures, response failures and arbitrary failures (van Steen and Tanenbaum, 2001). For simplicity and as an initial step towards tolerating failure of M_{sm} nodes in (Marufu et al., 2015), this article only considers crash failure. If a system cannot tolerate crash failures then there is no way it can tolerate the other classes of failure (Ghosh, 2014) (van Steen and Tanenbaum, 2001). A crash failure occurs when a process prematurely halts, but was working correctly until it stopped. We consider a typical crash failure of the M_{sm} nodes, where once the node has halted, nothing is heard from it anymore. Communication through M_{sm} is blocked; hence, the algorithm does not progress but instead will be held up until the node is recovered. The two failure situations to consider include crash of an M_{sm} node when in possession of the token or when not.

1. **If a Token Possessing M_{sm} Node Fails**: the algorithm in (Marufu et al., 2015) will not recover from failure of the site holding the token. They are different scenarios to be considered which include: when the token is being utilized by an M_{mp}, when the token is in transit between M_{mp}-M_{sm}, when the token is in the M_{sm} ready to be sent to another child M_{mp} node or when the token is in the M_{sm} ready to be passed to a requesting neighbouring M_{sm} node.
2. **Token Requesting M_{sm} Node Fails**: In this case, if the M_{sm} node responsible for forwarding the token request crashes, the path would have to be re-established. Failure of such a node results in partitioning of the network, leading to similar impacts

stated in the previous case.

4.2 Fault Tolerance Approach

As mentioned before, each M_{sm} node acts as a proxy to a cluster of M_{mp} nodes and its failure would need to be masked to allow for continued service to the child M_{mp} nodes. Thus, we introduce the concept of a redundant $M2_{sm}$ backup node to each M_{sm} node. The reason for opting for this approach is inspired by its simplicity and preservation of M_{sm} nodes' functionality thereby enabling trading agents hosted on M_{mp} child nodes to transact in the auction market. A downside to primary backup fault tolerance is that it handles Byzantine failures poorly because there is usually no check routine to make sure the primary is functioning correctly. In addition, the primary backup system must always be in agreement with the main system so that the backup node can take over the functions of primary node. Recovery from a primary node failure is usually time consuming and complex.

4.3 Fault Tolerant CDA Algorithm

We propose a fault tolerant distributed CDA algorithm. At the M_{sm} and $M2_{sm}$ nodes the algorithm executes the following routines: *GlobalTokenRequest*, *LocalTokenDistribution*, and *GlobalTokenTransfer*. The $M2_{sm}$ nodes also execute an additional routine called *CrashFailureHandling*. Contrary, M_{mp} nodes execute the following routines: *LocalTokenRequest*, *LocalMarketExecution* and *LocalTokenTransfer*. The fault tolerant CDA algorithm includes two types of request messages: Req_{ms} (global token request) and Req_{mp} (local token request). Incoming Req_{sm} from neighbouring M_{sm} nodes (including 'self') are enqueued in a FIFO $RQ1$ queue while Req_{mp} from child M_{mp} nodes are enqueued in a FIFO $RQ2$ queue. M_{sm} nodes have a $POINTER$ variable (points to the M_{sm} in possession of the token, or next intermediate M_{sm} pointing to a token holding node- see (Raymond, 1989)) where Req_{sm} is sent.

4.3.1 Local Token Request

When a *TA* needs to trade in the auction market, the *TA* triggers the hosting $M_{mp}i$ node to send a Req_{mp} to the M_{sm} provided that battery is above 10 percent and it does not already possess the token. When a Req_{mp} is sent to M_{sm}, the sleep mode is deactivated to ensure the M_{mp} node remains online to receive the token. On receipt of the token $FlagM_{mp}$ is turned to TRUE.

```
IF(FlagM_{mp} == FALSE)
  {IF(BatteryLife > 10%)
    { Send (ReqM_{mp};i) to M_{sm}
      Disable doze mode
      Wait until(FlagM_{mp}== TRUE)}}
ELSE
  Do not send ReqM_{mp}
```

4.3.2 Global Token Request

The *GlobalTokenRequest* routine will be the initial procedure that allows an M_{sm} node to send a request to participate in the mutual exclusion token exchange process. Once a Req_{sm} is sent to the neighbour node holding the token or in path to the token, *TokenAsked* which is a boolean variable is set to *TRUE*. This avoids forwarding of similar request messages to the same token holder. While the token is not received the token, $FlagM_{sm}$ remains FALSE. Two subroutines or methods namely *EnqueueRequest* and *Nodebackup* are executed while M_{sm} is waiting for the token. *EnqueueRequest* enqueues the Req_{sm} and Req_{mp} requests into *RQ1* and *RQ2* respectively, while *Nodebackup* creates a checkpoint by sending updates to the $M2_{sm}$ node.

```
IF (FlagM_{sm} == FALSE)
  {IF (TokenAsked == FALSE)
   {Send ReqM_{sm} to node in POINTER*
    Set TokenAsked to TRUE
    WHILE(FlagM_sm == FALSE)
       {IF (ReqM_{mp} == Received ||
            ReqM_{sm} == Received)
             {EnqueueRequest()
              NodeBackup() }
        ELSE
           Do nothing }
    Set FlagM_{sm} == TRUE }
   ELSEIF (TokenAsked==TRUE)
    WHILE (FlagM_sm == FALSE)
      {IF (ReqM_{mp} == Received ||
           ReqM_{sm} == Received)
            {EnqueueRequest()
             NodeBackup() }
       ELSE
          Do nothing   }
    Set FlagM_{sm} == TRUE
  }
```

4.3.3 Local Token Distribution

Once the token is received the algorithm moves to the *LocalTokenDistribution* routine. Receipt of the token by the requesting M_{sm} node sets the boolean variable $FlagM_{sm}$ to *TRUE* indicating possession of the token. *GQ* is a FIFO queue that stores a copy of requests submitted and "locked in" at the arrival of the token to the cluster head. Once a token is received all requests in *RQ2* are transferred to *GQ2*.

While there are still requests in *GQ*, the algorithm will run the *EnqueueRequest* and *BackupNode* subroutines before sending to an M_{mp} node with a request at the head of the *GQ*. After token is allocated to an M_{mp} node, the nodes corresponding Req_{mp} entry is removed from *GQ*. The M_{sm} node then waits for a predefined time for the return of the token. To ensure that M_{sm} does not wait for the token for an undefined time, the routine will consider a time bound on the wait period. Failure of the M_{mp} nodes to return the token within the predefined time results in the token degradation (an M_{mp}) and regenerated (at M_{sm}) from the last known backup.

```
IF (FlagM_sm == TRUE)
 {IF (self ReqM_{sm} is at head)
    {GQ <- RQ
     n <- GQ entries
     WHILE {n >= 1}
     { EnqueueRequest()
       NodeBackup()
       IF (M_{mp} at head != disconnected)
          {Assign token to $ M_{mp} $
           Remove from GQ
           Wait for token return
           IF (TokenReturn == TRUE)
               Send Token(TRUE) to next }
       ELSE
           {TokenReturn = timed-out
            TokenRegenerate() }
       n--} }
 ELSE
     Send Token(TRUE) to next  }
```

4.3.4 Global Token Transfer

A token-possessing M_{sm} node will send the token if it has a non-empty *RQ1* (where Req_{sm} at head of the *RQ1* is not its request). The boolean $FlagM_{sm}$ is then set to *FALSE* and the token is sent to the respective M_{sm} node with a Req_{sm} at the head of *RQ1*.

```
IF (FlagMsm = TRUE && RQ1 != empty &&
    ReqM_{sm} != 'self')
    {Set FlagM_{sm} to FALSE
     Send token to ReqM_{sm} at RQ1 head}
```

4.3.5 Local Market Execution

Once a token is received at the M_{mp} node, the *TokenCounter* variable is incremented. The *TA* is allowed to create an offer (bid/ask) which is submitted into the auction market order-book. *TokenOB* is an online copy of the CDA order-book in the token. *LocalOB* is a local copy of the CDA order-book updated each time a trading agent participates in the auction market. Each M_{mp} has a *ClusterDir* that contains a directory of neighbouring M_{mp} nodes. *TokenCounter* keeps a record of the number of auction market rounds. When a predefined number of rounds is reached trading is terminated. A message including trading day statistical data may be communicated to the rest of the participating nodes.

```
IF (FlagM_{mp} == TRUE)
   { TokenCounter++
     IF (tradeID == buyer)
       { FormOffer(ob, Pt, Pll, Pul)
         bid = offer
         IF (bid <= ob ||
             out of [P_{ll}, P_{ul}] range)
           bid is invalid
         ELSE
          { ob = bid
            IF (ob >= oa)
               P_{t} = oa
               Trade! and Obook update}  }
     ELSIF (tradeID == seller)
       { FormOffer(ob, Pt, Pll, Pul)
         ask = offer
         IF (ask >= oa OR
             out of [$ P_{ll}, P_{ul} $] range)
           ask is invalid
         ELSE
          { oa = ask
            IF (ob >= oa )
              P_{t} = ob
              Trade! and Obook update} }
     ELSE
      {no new oa/ob in pre-specified period
       Round ended with no transaction }
 Update LocalOB from TokenOB}
```

4.3.6 Local Token Transfer

Once a trading agent has completed its execution in the auction market the token is returned to M_{sm}. If the connection to M_{sm} is ALIVE the token is send back, if not the token-possessing M_{mp} tries to send the token via a neighbouring M_{mp} with probably the best connectivity to M_{sm}.

```
IF (Connection to M_{sm} == ALIVE)
    Return Token to M_{sm}
ELSIF (Connection to M_{mp} via M_{sm} == ALIVE)
    Return Token to M_{sm}
ELSE
  { Destroy the token
    Revert back changes in the LocalOB }
 Set FlagM_{mp} to FALSE
```

4.3.7 Crash Failure Handling

This routine is executed at the $M2_{sm}$ node. An in-depth discussion of the crash failure handling strategy executed in this routine is presented in Subsection 4.4.

```
IF (Timeout = TRUE OR N_Query = TRUE)
   { Ping M_{sm} with Enquiry() message
     IF (M_{sm} respond = TRUE)
        {M_{sm} is alive
         Wait for NodeBackup() }
     ELSE
        {FailureContainment()
        Resume M_{sm} operations } }
```

4.4 Crash Failure Handling Strategy

There are four general activities defined by (Jalote, 1994) that systems employing fault tolerance have to perform: error detection, failure containment, error recovery, and fault resolution and continued system service. These considerations have been factored into the new fault tolerant protocol.

4.4.1 Error Detection

This phase deduces the presence of a fault by detecting an error in the state of a subsystem. It is from the presence of errors that failures and faults can be deduced. Unless errors can be detected successfully by a fault model, the fault model is somewhat useless. We consider that detection of M_{sm} node failure is done by use of timing checks. In our context if the standby node $M2_{sm}$ does not receive a backup message from the M_{sm} node with respect to the timer clock, it will send a "query" message to the M_{sm}. If no response is received, it is assumed the M_{sm} has failed which triggers $M2_{sm}$ to enter the recovery phase (section 4.3.7). We consider that each $M2_{sm}$ carries a failure detector, to detect crashed M_{sm}. A failure detector is called strongly accurate if only crashed processes are ever suspected (Fokkink, 2013). We understand in bounded delay networks, a strongly accurate (and complete) failure detector is implemented as follows: Suppose l_{max} is a known upper bound on network latency from M_{sm} to $M2_{sm}$. Each M_{sm} thus broadcasts an "alive" message every ν time units. Each M_{sm} from which no message is received for $\nu + l_{max}$ time units is suspected to have crashed. An inquiry message is then send to confirm this suspicion in algorithm.

4.4.2 Failure Containment

The system design incorporates mechanisms to limit the spreading of errors in the system, thereby confining the failure to predetermined boundaries (line: *FailureContainment* in 4.3.7). In order to prevent the spread of error, the faulty node is suspended from any participation and its tasks are taken by the standby component. We assume this stand-by component does not share similar properties susceptible to error detected in the primary node.

4.4.3 Error Recovery

The two general techniques for error recovery are backward and forward recovery (Jalote, 1994). We chose backward recovery in which check-pointing is done frequently on the standby $M2_{sm}$ node and in the event of a failure a system roll-back occurs. One main drawback of the backward technique is the overhead required. Assuming the $M2_{sm}$ nodes have a fairly stable storage, frequent check-pointing is required which affects the normal execution of the system even in the absence of failures. Despite the high overhead, we opt for backward recovery due to its simplicity and independence from the nature of the fault or failure (Jalote, 1994). We assume check-pointing invokes the $NodeBackup$ procedure everytime a token is returned by M_{mp} to M_{sm}, or when a $ReqM_{mp}$ or a $ReqM_{sm}$ message is received. To reduce the overhead, one message (including M_{sm} current data structures) can be sent to the $M2_{sm}$ if M_{sm} is in possession of the token and is check-pointing. We assume a roll-back procedure is executed at the $M2_{sm}$ node with regards to the last checkpoint.

4.4.4 Fault Resolution and Continued Service

A fault tolerant system has to function such that the faulty components are not used through a fault containment method. By assuming the main component's role, the backup $M2_{sm}$ node masks primary M_{sm} node failure, isolating the faulty component but maintaining system availability and reliability. The $M2_{sm}$ reestablishes connection with M_{mp} nodes then resumes communication with the neighbouring M_{sm} nodes.

4.5 Proposed CDA Algorithm Analysis

We use the fault scenarios in analysing if the modified fault-tolerance algorithm is able to address the faults noted as well as maintain the crucial mutual exclusion properties. Further we discuss message and time complexity of the fault tolerant distributed CDA algorithm.

4.5.1 CDA Algorithm Correctness

We analyse how the fault tolerant CDA algorithm guarantees the following properties in the presence of potential fault, thus failure:

- ME1: If token-handling is diligently implemented only one token will be in the network at any particular time even in the presence of faults. The proposed algorithm ensures that at any instant of time, not more than one node holds the token. Whenever a node receives a token, it becomes exclusively privileged. Similarly, when a node sends the token, it becomes unprivileged. Between the instants there is no privileged node. Thus, there is at most one privileged node at any point of time in the network. If M_{sm} granting the token fails then a privileged cluster node M_{mp} will not be able to

return the token within a predefined time. The token session is cancelled and the token is considered lost. The same M_{mp} node will revert back the changes to the time before token was received and the timed out token is destroyed before a token regeneration procedure in the $M2_{sm}$ node. If a privileged M_{sm} node finds itself without outgoing links to neighbouring M_{sm} nodes and the timer has elapsed or if it has failed, it will destroy the copy of the token before failure on reboot or resumption of services. The respective $M2_{sm}$ backup node will regenerate the token with all details with respect to the last checkpoint. Thus, only one token can be in the system guaranteeing mutual exclusiveness.

- ME2: When the token is free, and one or more *TA*s want to enter the auction market but are not able to do so, a deadlock can occur. This happens due to
  - the token not being able to be transferred to a node because no node holds the privilege,
  - the node in possession of the token is unaware that there are other nodes requiring the privilege, or
  - the token fails to reach a requesting unprivileged node.

  We are aware that the logical pattern established using POINTER variables ensures a node that needs the token sends $ReqM_{sm}$ either to $M_{sm}v$ holding the token or to $M_{sm}u$ a neighbouring node that has a path to the token holder. M_{mp} nodes send $ReqM_{mp}$ to their cluster head M_{sm} node. If we consider the orientation of tree links formed by the M_{sm} nodes, say L at time t the resulting directed graph. In all cases, there are no directed cycles, making $L[t]$ acyclic, where from any non-terminal node there is a directed path to exactly one terminal entity. Within finite time, we consider every message will stop travelling at a M_{sm}. If a privileged node fails our algorithm ensures that the copy in the failed node is deleted while a copy is regenerated from the last checkpoint. By establishing connection with the neighbouring M_{sm} nodes and M_{mp} nodes previously connected to the primary the token path is complete and the $M2_{sm}$ will be aware which node to pass the token.
- ME3: If $M_{sm}u$ holds the token and another node $M_{sm}v$ requests for the token, the identity of $M_{sm}v$ or of proxy nodes for $M_{sm}v$ will be present in the *RQ1*s of various M_{sm}nodes in the path connecting the requesting node to the currently token-holding node. Thus depending on the position of the node $M_{sm}v$ requests in those *RQ1*s, $M_{sm}v$ will sooner or later receive the privilege. In addition, enqueuing $ReqM_{mp}$ requests in $RQ2$ ensures that a token gets released to the next M_{sm} at the head of the $RQ1$ once the GQ is empty, no single cluster of M_{mp} nodes continues to trade while other nodes are starved. In the presence of failure the static spanning tree may be partitioned, which means a sub-tree that has the token will benefit while the other sub-tree(s) is starved of the token. Our algorithm ensures that partitioning does not occur with a backup node taking the place of the failed node.
- ME4: Every trading agent has the same opportunity to enter into the auction market. The *FIFO* queues *RQ1* and *RQ2* are serviced in the order of arrival of the requests from nodes. A new request from an agent that acquired the token in the recent past is enqueued behind the remaining pending requests. In a situation that a privileged M_{sm} node fails while granting the local cluster M_{mp} nodes a chance to participate in the auction, our algorithm ensures the remaining M_{mp} nodes get a chance to obtain the token and transact in the order in which they send their requests for the token. By allowing the backup $M2_{sm}$ to resume the roles of the failed M_{sm} instead of eliminating the failed M_{sm} node, fair token distribution is maintained.

4.5.2 CDA Algorithm Efficiency

Runtime: Usually it suffices to identify a dominant operation and to estimate the number of times it is executed. In the proposed Algorithm we consider $LocalTokenDistribution$ (executed on M_{sm})to be the most costly operation carried out. As a dominant operation we regard the while loop (line 22). This running time depends linearly on the input's size i.

Time Complexity: Analysis of the running time of our fault tolerant CDA algorithm may not be sufficient as we are more interested in analysing how this running time increases when the input size increases. Thus we consider the *order of growth* measure which can be estimated by taking into account the dominant term of the running time expression. The dominant operation (section 4.3.7) is influenced by N which is the number of cluster nodes. As n grows the time complexity increases linearly at an order of $O(N)$.

Message Complexity: Assuming that $V = \sum(v_1 + v_2 ... + v_n)$ is the number of Req_{sm} received by a non-token-possessing node $M_{sm}i$ from its neighbours because it is in line to a token holding node. While $U = \sum(u_1 + u_2 + ... + u_n)$ is the number Req_{mp} received by $M_{sm}i$. Each time a $BackupNode$ is

executed W, which is the number of check-pointing messages: $W = \sum(U + V)$, is saved in the stable $M2_{sm}$ storage. In a worst case scenario all the N cluster M_{mp} nodes will send a Rep_{mp} at random times to M_{sm} while all child M_{sm} nodes (in a K-ary tree) send Req_{sm}. This results in a message complexity of $O(W)$.

5 CONCLUSIONS

We proposed a fault tolerant distributed CDA algorithm for energy allocation within computationally constrained microgrids. The fault tolerance proposed specifically addresses crash faults. Instead of bypassing the failed node and reconnecting the remaining spanning tree segments, our algorithm masks failure by incorporating redundancy to each cluster head node. The fault tolerant CDA allows trading agents to have mutually exclusive permission to participate in the auction market despite the presence of crash failures. We present the correctness and efficiency of the fault tolerant algorithm. The runtime of the algorithm is linearly dependent on the input size i. The time complexity for a single agent (hosted on a cluster node) to trade in the auction market is $O(N)$ where N is number of cluster nodes; and the message complexity is $O(W)$, where W is the number of check-pointing messages; N is the number messages exchanged per critical section execution. These are reasonable upper bounds of a fault tolerant supported CDA algorithm employing redundancy and check-pointing. While minimal redundancy is likely to be appropriate for the microgrid case, as future work we seeks to make a comparative evaluation to other schemes that might be expensive for small amounts of extra fault-tolerance, but whose costs are lower when higher degrees of fault tolerance are desired. In distant future work we seek to investigate how malware may affect fairness, thus stability of a fault tolerant CDA augmented microgrid.

ACKNOWLEDGEMENTS

This work was done with the joint SANCOOP programme of the Norwegian Research Council and the South African National Research Foundation under NRF grant 237817; the Hasso-Plattner-Institute at UCT; and UCT postgraduate funding.

REFERENCES

Bloch, A. (2003). *Murphy's law*. Penguin.

Borenstein, S., Jaske, M., and Rosenfeld, A. (2002). Dynamic pricing, advanced metering, and demand response in electricity markets. *Center for the Study of Energy Markets*.

Chang, Y.-I., Singhal, M., and Liu, M. T. (1990). A fault tolerant algorithm for distributed mutual exclusion. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*, pages 146–154. IEEE.

Cui, T., Wang, Y., Nazarian, S., and Pedram, M. (2014). An Electricity Trade Model for Microgrid Communities in Smart Grid. In *Innovative Smart Grid Technologies Conference (ISGT), 2014 IEEE PES*, pages 1–5. IEEE.

Dhamdhere, D. M. and Kulkarni, S. S. (1994). A token based k-resilient mutual exclusion algorithm for distributed systems. *Information Processing Letters*, 50(3):151–157.

Fokkink, W. (2013). *Distributed Algorithms: An Intuitive Approach*. MIT Press.

Garg, V. K. (2011). *Principles of distributed systems*. Springer Publishing Company, Incorporated.

Ghosh, S. (2014). *Distributed systems: an algorithmic approach*. CRC press.

Izakian, H., Abraham, A., and Ladani, B. T. (2010). An auction method for Resource Allocation in Computational Grids. *Future Generation Computer Systems*, 26(2):228–235.

Jalote, P. (1994). *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc.

Kshemkalyani, A. D. and Singhal, M. (2008). *Distributed computing: Principles, Algorithms, and Systems*. Cambridge University Press.

Marufu, A. M., Kayem, A. V., and Wothulsen, S. (2015). A distributed continuous double auction framework for resource constrained microgrids. In *Critical Information Infrastructures Security, 2015. The 10th International Conference on*, pages –. IEEE.

Médard, M. and Lumetta, S. S. (2003). Network reliability and fault tolerance. *Encyclopedia of Telecommunications*.

Pałka, P., Radziszewska, W., and Nahorski, Z. (2012). Balancing electric power in a microgrid via programmable agents auctions. *Control and Cybernetics*, 41.

Raymond, K. (1989). A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7(1):61–77.

Raynal, M. (1986). Algorithms for mutual exclusion.

Revannaswamy, V. and Bhatt, P. (1997). A fault tolerant protocol as an extension to a distributed mutual exclusion algorithm. In *Parallel and Distributed Systems, 1997. Proceedings., 1997 International Conference on*, pages 730–735. IEEE.

Stańczak, J., Radziszewska, W., and Nahorski, Z. (2015). Dynamic Pricing and Balancing Mechanism for a Microgrid Electricity Market. In *Intelligent Systems' 2014*, pages 793–806. Springer.

Tanenbaum, A. S. and Van Steen, M. (2007). *Distributed Systems*. Prentice-Hall.

van Steen, M. and Tanenbaum, A. (2001). Distributed systems, principles and paradigms. *Vrije Universiteit Amsterdam, Holland*, pages 1–2.

Walter, J. E., Welch, J. L., and Vaidya, N. H. (2001). A mutual exclusion algorithm for ad hoc mobile networks. *Wireless Networks*, 7(6):585–600.