# Performance Evaluation of HARP: a Hierarchical Asynchronous Replication Protocol for Large Scale Systems

*Noha Adly*
Computer Laboratory
Cambridge University
Cambridge CB2 3QG, UK.

**Abstract**

*This report evaluates the performance of* HARP*, a hierarchical replication protocol based on nodes organised into a logical hierarchy. The scheme is based on communication with nearby replicas and scales well for thousands of replicas. It proposes a new service interface that provides different levels of asynchrony, allowing strong consistency and weak consistency to be integrated into the same framework. Further, it provides the ability to offer different levels of staleness, by querying from different levels of the hierarchy. We present results from a detailed simulation analysis evaluating the benefits and losses in performance resulting from using synchronous versus asynchronous operations within* HARP *under different system configurations and load mixes. Further, the performance is evaluated on different network topologies. An analytical solution based on the Open Queueing Network Model with Multiple Job Classes is carried out for the verification of the simulation model and the results are presented.*

## 1 Introduction

Data is replicated in distributed systems to improve system availability and performance. In recent years, the growth of internetworks and distributed applications has increased the need for large scale replicated systems. However, existing replication protocols do not address the scale and autonomy issues adequately. An internetwork consists of a set of local networks, connected by point-to-point links and gateways or routers. Communication with local replicas is usually much faster than accessing replicas on an internetwork. Therefore, protocols based on communication will *all* or distant replicas should be avoided in internetworks.

There are two approaches for managing replicated data: strong and weak consistency approaches. *Strong consistency* protocols enforce strict serializability by means of quorums [Gif79, CAA90, Kum91] and require *synchronisation* among a large number of replicas in order to ensure that replicas are mutually consistent. Synchronisation across internetworks is inappropriate for large scale systems. They suffer from relatively high latency and low throughput and a large number of replicas generate considerable traffic over the network.

In contrast, *weak consistency* protocols provide higher availability and better response time by allowing updates and queries to occur *asynchronously* at any replica. A weak consistency approach should provide a propagation mechanism which ensures that updates are efficiently

and reliably propagated to all replicas even if the communication network does not provide such a guarantee. However, this approach is based on the assumption that the applications can tolerate some inconsistency and reconciliation methods should be available to resolve conflicts. Typical applications that have used weak consistency are Internet news, air traffic control, airline reservation and stock exchanges. Grapevine [SBN84] and the Global Name Service [Lam86] were among the first systems to use weak consistency. Other weak consistency protocols were presented in [DGH+87, DGP90, Gol92, LLS90, QP93, WB84]. These protocols are useful and interesting; however, they require a node to communicate with *all* other nodes. This would be acceptable for small networks with a few replicas, but it will incur a large communication overhead for wide area networks like the Internet.

This paper evaluates the performance of HARP, a hierarchical replication protocol that scales well for thousands of replicas while ensuring reliable delivery. It is based on communication with nearby replicas, providing low access latencies and limiting message traffic. The protocol provides an efficient propagation scheme that is based on replicas organised into a logical hierarchy where each node has to communicate with a few nodes only (in the local proximity). Further, it proposes a novel service interface that provides *asynchronous operations* that commit at any replica then propagate to other replicas and *synchronous operations* that commit after a quorum is assembled from nodes of the top level of the hierarchy then propagate downward. This interface gives an application the ability to select strong or weak consistency semantics. Synchronous operations can scale better than traditional approaches since HARP assembles quorums from a subset of the replicas rather than from all replicas. Due to the hierarchical pattern of propagation, the scheme provides the ability to locate replicas that are more up-to-date than others, depending upon the needs of various applications.

In this paper we evaluate the benefits and losses in performance resulting from using synchronous versus asynchronous operations under different system configurations and load mixes. Further, the performance is evaluated over different network topologies to see how they affect performance. The paper is organised as follows. The next section gives a brief description of the propagation scheme and interface of HARP. Section 3 describes the details of the simulation model. Section 4 presents the conducted experiments and results. Section 5 describes an analytical solution of the suggested model for verifying the simulator and presents its results. Finally, section 6 concludes the paper.

## 2 Protocol description

In HARP, the scheme is based on organising the replicas into a logical, multilevel hierarchy. In this hierarchy, nodes are grouped into clusters, and clusters are organised into a tree, such that each cluster is assigned a parent node in its parent cluster. Nodes in the same cluster should have efficient communication between them as well as with their parent.

### 2.1 The propagation scheme

The propagation scheme is responsible for spreading updates to every node in the network while ensuring reliable delivery. The algorithm works as follows: a node $i$, originating a message, assigns it a unique identifier, sends the message to its neighbours, parent and children, and then waits for their acknowledgements (timeouts and retransmissions are used to overcome lost

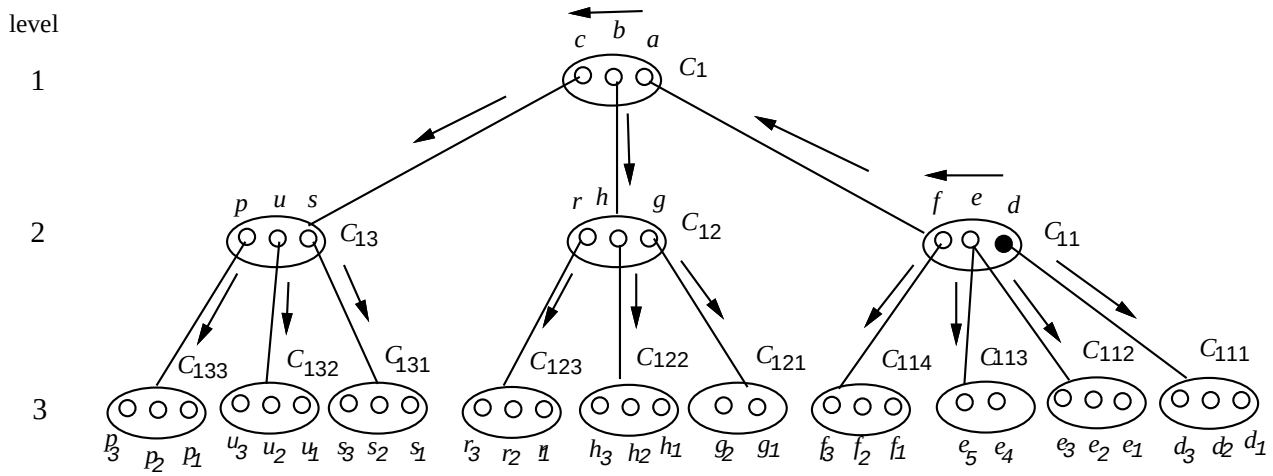


Figure 1: 40 nodes organised in a hierarchy of three levels

messages). Each receiving node $j$ sends an acknowledgement to the sender, then it passes the message to the next level as follows: if the message is coming from a neighbour or from the parent, then $j$ sends the message to its children; else, if it is coming from a child then $j$ sends it to its neighbours, its parent and to its children of clusters other than the one the message is coming from. This works recursively and a message originating at any site will eventually propagate everywhere. Figure 1 shows 40 nodes organised into a hierarchy of three levels with the path that a message would follow while propagating if it originated at node $d$. Details of the algorithm, restructuring operations to build and reconfigure the hierarchy dynamically as well as proofs of correctness can be found in [Adl93, ANB93].

### 2.2 Synchronous and asynchronous operations

Different applications require different degrees of consistency. HARP supports a set of operations that provide different levels of asynchrony allowing strong and weak consistency to be integrated into the same framework. Hence, it gives the application the flexibility to tailor the service to achieve the required degree of consistency by choosing the appropriate operations to manage its data. The following operations are supported:

- Fast_Write: a Fast_Write can be initiated and committed at any replica (the site of origin), and then propagated to other replicas using the propagation scheme.
- Fast_Read: returns the value read from any replica. It is associated with a parameter $x$, denoting the node to read from. The default of $x$ is the local node.
- Slow_Write: a Slow_Write can be initiated by any replica; but does not commit until a quorum has been assembled from the *root* nodes at the top of the hierarchy. For node $x$ to perform a Slow_Write, it sends the message to its ancestor node $y$ in the top cluster. Node $y$ assembles a quorum from its neighbours (using majority quorum consensus or any other standard method). Once the nodes of the top cluster commit the update, they propagate it down the tree and the initiating node commits the update on receipt of the commit message.

- Slow_Read: returns the value read from a quorum of nodes of the top cluster. A node $x$, wishing to perform a Slow_Read, sends the request to its ancestor node $y$ in the top level. Node $y$ assembles a quorum from its neighbours and returns the read value to $x$.

- Opt_Write: is similar to Slow_Write, but it is applied to the database of the site of origin and, optionally, to some other selected replicas (called *optimistic nodes*) before commitment. However, these updates are subject to being *undone* if while assembling the quorum at the top level, the decision was to abort the update. Then, every replica that has *done* the update must *undo* it.

Slow_Write should be used by operations that are likely to result in conflict and where no conflict reconciliation is acceptable, such as some banking applications. Slow_Read, if used in conjunction with Slow_Write, will return the most up-to-date value. Opt_Write can be used by operations that conflict with very low probability; however, if conflicts occur their semantics can accept an abort, such as airline reservation. Also, it can be used instead of Slow_Write if a partition occurs, allowing operations to proceed; however, again, they should be ready to abort when the partition heals in case conflicts occur. If an application restricts all its operations to Slow_Write and Slow_Read, then *strong consistency* (strict serializability) is achieved. It should be noticed that, by assigning the top cluster an adequate number of nodes, this scheme will result in better availability than the primary copy method and better latency and communication overhead than the usual quorum consensus methods since the number of nodes to synchronise with is small. Applications that require a lower degree of consistency i.e. *weak consistency*, such as stock exchange or bibliographic databases, can use Fast_Write, Fast_Read and Opt_Write.

Due to the hierarchical pattern of propagation that the updates follow, the scheme provides the ability to locate replicas that are more up-to-date than others while propagation is in progress. Nodes in a high level of the hierarchy usually offer better information than nodes in low levels. Therefore, Fast_Read can be used to provide different levels of staleness, depending upon the needs of various applications, by reading from replicas at different positions in the hierarchy.

# 3 The simulation model

A simulation model was designed and implemented for performance evaluation of the proposed protocol. This section presents the design of the simulation model. It describes the system and the network models, the overhead of the algorithms and the way the various operations are modelled. Then, it defines the performance metrics used for evaluation. The model is described by a set of parameters which are listed in Table 1.

The simulator was implemented using the simulation package SIMPACK [Fis92] developed at the University of Florida and based on the C programming language.

## 3.1 The System Model

The model consists of a number nodes connected by a communication network. Each node is assumed to store a copy of the object being replicated. A node has nonvolatile storage (disk) and a CPU. CPUs are modelled as single queue with single server; with no preemption. Each disk has its own queue, which it serves in FIFO manner.

| *parameter* | Description | Default value |
|---|---|---|
| $n$ | Number of nodes in the network | 12 |
| *inter_arr_rate* | Inter arrival rate | 1000 msec |
| $freq_r$ | Frequency of read requests | .85 |
| $fast_{r0}$ | Frequency of Fast_Read_0 requests | var [0-1] |
| $fast_{r1}$ | Frequency of Fast_Read_1 requests | var [0-1] |
| $slow_r$ | Frequency of Slow_Read requests | var [0-1] |
| $fast_w$ | Frequency of Fast_Write requests | var [0-1] |
| $slow_w$ | Frequency of Slow_Write requests | var [0-1] |
| $opt_w$ | Frequency of Opt_Write requests | var [0-1] |
| *cpu_req* | Processing requirements of a request (exponential) | 50 msec |
| *io_req* | IO requirements of a request (exponential) | 60 msec |
| *cpu_msg* | CPU overhead to send/receive a message | 2.0 msec |
| *lan_del* | Network delay over LAN (uniform) | 0.02 msec |
| *wan_del* | Network delay over WAN (exponential) | 10.0 msec |
| *prop_ovhd_const* | Propagation overhead: constant part | 2.0 msec |
| *prop_ovhd_var* | Propagation overhead: variable part (per destination) | 1.0 msec |
| *pc_ovhd_const* | 2PC overhead: constant part | 2.0 msec |
| *pc_ovhd_var* | 2PC overhead: variable part (per participant) | 1.0 msec |
| *cpu_lock* | CPU time to acquire/release a lock | 0.2 msec |
| *deadlock_timeout* | Timeout signalling a deadlock detection | adaptive |
| *backoff_per* | Time interval an aborted operation waits before restarting | adaptive |
| *cpu_abort* | Overhead of aborting an operation | 2.0 msec |
| *pr_commit* | Probability an Opt_Write operation will commit | 0.9 |
| *cpu_undo* | CPU time to undo an aborted operation | 50 msec |
| *io_undo* | IO time to undo an aborted operation | 60 msec |
| $power_cpu_i$ | Speed of CPU at node $i$ | 1 |
| $power_io_i$ | Speed of IO at node $i$ | 1 |
| $power_transc_i$ | Speed of transceiver $i$ | 1 |
| *batch_length* | Length of a single simulation batch | 2000 job |
| *num_batch* | Number of batches for which simulation runs | 3 |
| *warmup_per* | Number of jobs after which statistics are reset | 200 job |

Table 1: System parameters

The system is modelled as an open system where requests arrive with arrival rate drawn from a Poisson distribution. Each request is composed of a single operation (Read or Write). Each operation is parametrised by *cpu_req* and *io_req* specifying the average amount of CPU and IO time required respectively; they are exponentially distributed. The fraction of read requests is determined by the parameter $freq_r$. In this study, two Fast_Read operations are evaluated to reflect the effect of reading from different levels of the hierarchy. The first operation, denoted by Fast_Read_0, reads from the local node; and the second one, denoted by Fast_Read_1, reads from a node at one level up in the hierarchy, that is from the parent node. The parameters $fast_{r0}$, $fast_{r1}$ and $slow_r$ specify the frequency of the Fast_Read_0, Fast_Read_1 and Slow_Read operations respectively. The parameters $fast_w$, $slow_w$ and $opt_w$ specify the frequency of Fast_Write, Slow_Write and Opt_Write operations respectively. These parameters allow the examination of a wide variation of workload mixes.

### 3.2 The Network Model

The physical network modelled was chosen to reflect a hierarchical structure, as shown in Figure 2. It consists of 12 nodes, three of them are fully connected through long distance links to form a backbone. The remaining nodes are grouped into three-clusters. Within a cluster, nodes are fully connected through high speed links (LAN) and experience very low delay in sending messages to each other. Each cluster is connected to a node on the backbone through a long distance link. This is similar to the structure of the Internet today: regions of high connectivity with regions weakly connected through a backbone. The logical hierarchy was chosen to match the physical hierarchy; that is it consists of two levels, nodes on the backbone form one cluster at level one and each LAN cluster forms a logical cluster at level two.

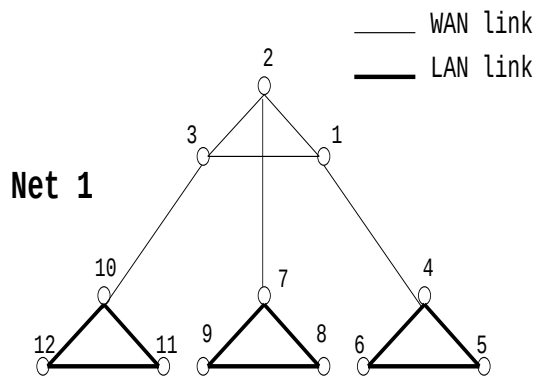


Figure 2: The physical network modelled, Net 1

Messages are reliably delivered and FIFO channels are assumed. The system is modelled during normal operation; that is, in the no failure case.

Sending messages between nodes involves the overhead of running the network protocols and the operating system. This overhead is modelled as *cpu_msg* msec processing at the CPU. A node performs this overhead when sending a message, receiving a message or relaying a message as an intermediate node between a sender and a destination node. It is assumed that there is a

multicast facility rather than point-to-point communication. Also messages encounter network delays which involve transmission delays, propagation delays and media access time. Delay over a LAN is modelled as *lan_del* msec drawn from a uniform distribution and transceivers are modelled as single queue with an infinite number of servers. For WAN links this delay is modelled as *wan_del* msec, exponentially distributed, and transceivers are modelled as single server single queue. When a node $i$ sends a message to a node $j$, this involves communication processing overhead (*cpu_msg*) at $i, j$ and at every intermediate node on the route from $i$ to $j$. Further, the message encounters network delays at each link (WAN or LAN) it crosses.

### 3.3 Overhead of the Algorithms

The **propagation algorithm** is modelled as a constant overhead (*propg_ovhd_const*), which accounts for the time to check for duplicates, update status, decide from whom a message came and to whom to send it to. Further, an overhead is added for every logical destination it is sent to (*propg_ovhd_var*), which accounts for the time to maintain timers, handle acknowledgements and so on.

For Slow_Write, Slow_Read and Opt_Write, the generic **Two-Phase-Commit algorithm** (2PC) [BHG87] is adopted while collecting quorums. The overhead of 2PC at the coordinator CO (for phase 1 and 2) is modelled as a constant overhead (*pc_ovhd_const*) plus an overhead for each node participating in the quorum (*pc_ovhd_var*). The overhead of acquiring and releasing locks is modelled as *cpu_lock* msec processing at the CPU. The overhead of writing on logs is ignored. The simulator maintains lock tables and explicitly simulates lock contention.

**Deadlock avoidance** is implemented by using a timeout interval based on the following heuristic [CLO87]: *deadlock_timeout = avg(W) + k * σ(W)*, where *avg(W)* is the average waiting time of lock requests, *σ(W)* is the standard deviation of waiting time of lock requests and $k$ is a weighting factor. Hence, *deadlock_timeout* is dynamically adjusted to reflect on-line estimation of lock request time. $k$ was taken to be 1 as recommended by [CLO87]. When deadlock is detected, each node releases held locks and aborts the operation. Aborting an operation is modelled as processing time overhead (*cpu_abort*) at CO and each participant involved in the quorum. A *backoff_period*= $\alpha *$ *deadlock_timeout* is used, then the operation is restarted.

**Majority quorum consensus** has been adopted to collect quorums from the top cluster for synchronous operations, although any other policy could have been used. Several **quorum collection policies** are discussed in the literature [AR91, CK94, GL92]. Two straightforward solutions are to poll nodes serially one at a time until $q$ nodes have responded (where $q$ is the quorum size) or to send the request in parallel to *all* nodes. The first solution experiences excessive delay while the second incurs high traffic and causes congestion. Other policies [GL92] suggested sending to $q$ nodes in parallel then contacting more if report failures occur or to send to $q + \epsilon$ nodes in parallel. Since the extra nodes to send to are to account for failures, and since in our model it is assumed that failures do not occur, the adopted policy is to send the request to exactly $q$ nodes in parallel. In [CK94], several heuristics have been presented to access $q$ nodes out of $m$ with minimum costs given the topology and communication cost between each pair of nodes. However, since nodes in the top cluster are assumed to be fully connected with the same link speeds, then random selection is adopted.

### 3.4 Operation Modelling

In the following, a description of how each operation is modelled in the simulator is presented. To maintain the version of replicas, the simulator keeps a global data version counter, denoted by *global_version*, and a version counter for each node $j$, denoted by $version_j$.

Fast_Write : A node $i$ originating a Fast_Write executes the request and updates versions by incrementing the *global_version* and its own version $version_i$. Then, it sends the message to its correspondents according to the propagation algorithm. This involves processing the overhead of the propagation algorithm as well as communication overhead. When the message reaches a destination $j$, $j$ sends the message to its correspondents (which again involves propagation and communication overhead). Then, it executes the request and updates its own version $version_j$. The same procedure is repeated until the update reaches all leaf nodes. The updates are delivered at the replicas in any order.

Fast_Read_0 : this operation just involves executing the request at the origin node.

Fast_Read_1 : A node $i$ originating a Fast_Read_1, sends the read request to its parent $p$. When the message reaches $p$, it executes the read, then it sends back the results to $i$. When $i$ receives the message, the read is considered complete.

Slow_Write : A node $i$ originating a Slow_Write, if $i$ is not in the top cluster, then it sends the update request to its ancestor in the top cluster, which acts as the *coordinator* (CO) of the operation; otherwise, $i$ acts as its own CO. CO, receiving the message, starts processing phase one of 2PC, acquires locks and sends vote_req message to (quorum size -1) nodes of its neighbours selected randomly. A *participant* receiving vote_req, acquires locks; when obtained it sends vote_yes to CO. When CO receives vote_yes from all participants, it executes the overhead of phase two. At this point the update has committed and CO sends commit message to the participants and the origin node $i$. Then, it executes the update, updates the *global_version* and its own version and releases locks. Next, it starts propagating the message by sending it to neighbours that did not participate in the quorum and to children other than $i$. A participant receiving a commit, executes the update, updates its version, releases locks and propagates the message to its children. Node $i$ receiving commit, executes the update, updates $version_i$, returns control to the user then propagates the message to its children. Any other node receiving the message, propagates the message to its children then executes the update and updates its version. If CO times out before a quorum is assembled (a deadlock has occurred), then it sends abort message to all participants, executes the abort protocol and releases locks. It waits for *backoff_per* then restarts. A participant receiving an abort, executes the abort protocol and releases locks.

Slow_Read : The same steps as Slow_Write are followed until CO decides to commit then executes phase two of 2PC. Then, it checks if it has the highest version among the quorum. If it does, then it sends commit to all participants, executes the read request, releases locks and sends results to the origin node. A participant receiving a commit releases locks. If CO is not the most up-to-date copy, it selects one of the participants that has the highest version, sends it an exec_read message, and sends commit to the remaining participants.

A participant receiving an exec_read, executes the read request, releases locks and sends back results to CO which will forward them to the origin node.

Opt_Write : Opt_Write originating at top cluster nodes, are exactly the same as Slow_Write. For Opt_Write originating at any other node $i$, $i$ sends the request to CO, executes the update and returns control to the user. In this study, nodes that are on the path from $i$ to CO are selected to be the *optimistic nodes*; that is, they execute the update as well before CO commits it. When the message reaches CO, it executes 2PC as in Slow_Write. When CO assembles the quorum, it decides whether the update will commit or abort according to an input parameter *pr_commit*. If the decision is to commit, then CO sends commit to all participants, origin node and optimistic nodes, executes the update, updates its version, releases locks and propagates updates to children excluding $i$ and all optimistic nodes. An origin or optimistic node receiving commit, propagates the message to children, excluding any optimistic node. A participant receiving a commit, executes the update, updates its version, releases locks and propagates the message to its children. If the decision was to abort, then CO sends abort to all participants, the origin node and optimistic nodes, executes the abort protocol and releases locks. A participant receiving abort executes the abort protocol and releases locks. The origin or an optimistic node receiving abort, undo the operation.

### 3.5 Parameter Setting

Parameter settings are a difficult issue in simulating a distributed system. Our parameters for the database and requests are chosen to be comparable to previous simulation studies such as [CDY90, CL91]. The frequencies of different classes of requests vary from experiment to experiment to generate a variety of load mixes. The values of the parameters relating to the 2PC protocol are in accordance with [CDY90].

The number of nodes is kept small for the tractability of the simulation. A single point in the experiments with 12-nodes networks required a run time on Alpha machines ranging from two to five hours, depending on the load mix, while with 39 nodes it required a run time of eight to 20 hours.

The communication overhead (processing overhead and delay on links) are in keeping with [Gra88, AS92, Zha94] and with a study on measurements of wide area networks presented in [PKL91]. It is based on a Layered Refinement methodology which measures performance characteristics on a live Internet and build an overall profile of end-to-end performance by collecting data and measurement of different layers such as application, operating system and network layers. The delay over a LAN is taken to be 0.02 msec split equally between propagation delay and transmission delay (assuming a message size of 1 Kbyte and a bandwidth of 1 Gbps). The delay over a WAN is taken to be 10 msec split equally between propagation delay and transmission delay (assuming a message size of 1 Kbyte and a bandwidth of 2 Mbps).

For many of the costs, the absolute values of the parameters are not particularly important; the key is their magnitude relative to the other parameters. The default values used for the system parameters are listed in Table 1.

### 3.6 Performance Metrics

The following performance metrics are considered:

- *response time*, the time from when the request is initiated until it is completed,
- *commit time*, the time it takes CO to commit a Slow_Write, Slow_Read or Opt_Write,
- *utilisation* of the resources,
- *throughput* of the system.

In order to evaluate the speed of propagation of updates, two metrics are introduced:

- *reach time*, the time at which an update reaches all replicas, and
- $coverage_i$, the time at which an update originating at any node has reached $i$ nodes.

In order to quantify the staleness of information read by a Fast_Read operation performed at node $j$, a new metric is introduced, denoted by $age_j$, which is defined as the number of updates that a read has missed. Therefore, the higher the value, the more stale the data is. This metric is computed as:

- $age_j = global_version - version_j + \gamma * abort_age_j$, where

$abort_age_j$ is the number of Opt_Write operations that $j$ has executed but that will abort. Since an optimistic node executes the update before it commits, then if a read occurs meanwhile, its age (i.e.number of missing updates) would be less if the update will eventually commit and will be higher if it will eventually abort. It is assumed that the penalty of reading from a node that has executed $x$ Opt_Writes which will abort, will add $x$ units to its age multiplied by a weighting factor $\gamma$. $\gamma$ is taken here to be 1. Therefore, in the simulation, when a node $i$ executes an Opt_Write that will eventually commit, it increments $version_i$. When a node $i$ executes an Opt_Write that will eventually abort, it increments $abort_age_i$ which will increase its age. When it receives the corresponding abort, and has undone the update, it decrements $abort_age_i$, which will restore its age and cancel the effect of having executed an update which will abort.

Also, a metric which is of interest is the *local age* which measures the number of updates missing between a node and other nodes in its neighbourhood rather than the global age. This metric is computed by taking the average of the differences between the version of the node performing the read and the versions of its neighbours, parent and children.

In the experiments, the means of the desired measurements are obtained by using the methods of batch means [Fer78]. *Num_batch* batches have been done for each run. Each run is left to run until at least *batch_length* jobs are generated at each node in the network. To eliminate the warmup effect, after generating *warmup_per* jobs at every node, all the statistics are reset. The results reported here are within the 90% confidence interval for the quantities measured. The measurements were within 10% of their true mean in almost all cases.

## 4 Experiments and results

In this section we present our performance results under several parameter variations such as read and write mixes, load intensity, communication overhead and network topologies.

The experiments are arranged into six groups. The first group (Section 4.1) considers synchronous updates only (i.e. Slow_Write and Opt_Write) and evaluates the performance of mixing synchronous reads (i.e. Slow_Read) and asynchronous reads (i.e. Fast_Read) under different system configurations. They are performed on the network Net 1 shown in Figure 2. The second group presents the performance over different network topologies when updates are synchronous

and the read mix is varied. The next two groups are similar to the first two but considering asynchronous updates (Fast_Write). The last two groups compare the performance over the different topologies of a 39-nodes network.

### 4.1 Experiments with Slow_Write and Opt_Write

In this section, several experiments are presented where updates are either Slow_Write or Opt_Write and reads are a mix of synchronous and asynchronous operations. It presents the results obtained under several parameter variations, namely, varying the read mix, the mix between Opt_Write and Slow_Write, the load intensity and the communication overhead. Those experiments are performed over the physical network Net 1, shown in Figure 2.

#### 4.1.1 Varying read mixes, when all writes are Slow_Write

Three experiments are carried out to explore the effect of varying a mix of Slow_Read and Fast_Read when all updates are Slow_Write. Experiment A, does not include Fast_Read_1 operations and varies the mix between Fast_Read_0 and Slow_Read. Experiment B, does not include Fast_Read_0 operations and varies the mix between Fast_Read_1 and Slow_Read. Experiment C varies the mix between Fast_Read and Slow_Read operations such that Fast_Read operations are split evenly between Fast_Read_0 and Fast_Read_1. The setting of the parameters is shown below; the remaining parameters are at their default values (Table 1).

| Exp | $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ |
|---|---|---|---|---|---|---|
| A | 1 | 0 | 0 | 0-1 | 0 | 0-1 |
| B | 1 | 0 | 0 | 0 | 0-1 | 0-1 |
| C | 1 | 0 | 0 | 0-1* | 0-1* | 0-1 |

* The ratio between $fast_{r0}$ to $fast_{r1}$ is kept 1:1

Results are plotted in Figure 3. We notice that, as $fast_r$ increases, the average response time of reads decreases substantially, whether fast reads are Fast_Read_0, Fast_Read_1 or a combination of both (Figure 3(a)). For instance, for a point where $slow_r$=0.5, there is an improvement over $slow_r$=1 of 56% when all fast reads are Fast_Read_0 (Exp A), 41% when all fast reads are Fast_Read_1 (Exp B) and 48% when fast reads are combination of both (Exp C). Fast_Read_0 has the lowest response time as it is executed locally, Fast_Read_1 has higher response time as it involves sending and receiving requests from the parent. Similarly, the response time of Slow_Write decreases as $fast_r$ increases as shown in Figure 3(b). The improvement is of 37% (Exp A), 24% (Exp B) and 31% (Exp C) for $slow_r$=0.5). Slow_Write associated with Fast_Read_0 (Exp A) experiences lower response than when associated with Fast_Read_1 (Exp B) and it lies in between with a mixture of Fast_Read_0 and Fast_Read_1 (Exp C).

This reduction in the response time of reads and writes is basically due to three reasons:

1. The load on the resources of the nodes in the top level reduces as Fast_Reads are introduced: these nodes perform less overhead of 2PC, and are exempted from performing the read requests, which are done locally. Therefore, the load is more balanced on the resources.

2. The traffic on the communication links decreases, especially in the presence of Fast_Read_0 as they require no communication.

3. The data contention (lock contention) decreases as $slow_r$ decreases, which improves the commit time of Slow_Read and Slow_Write.

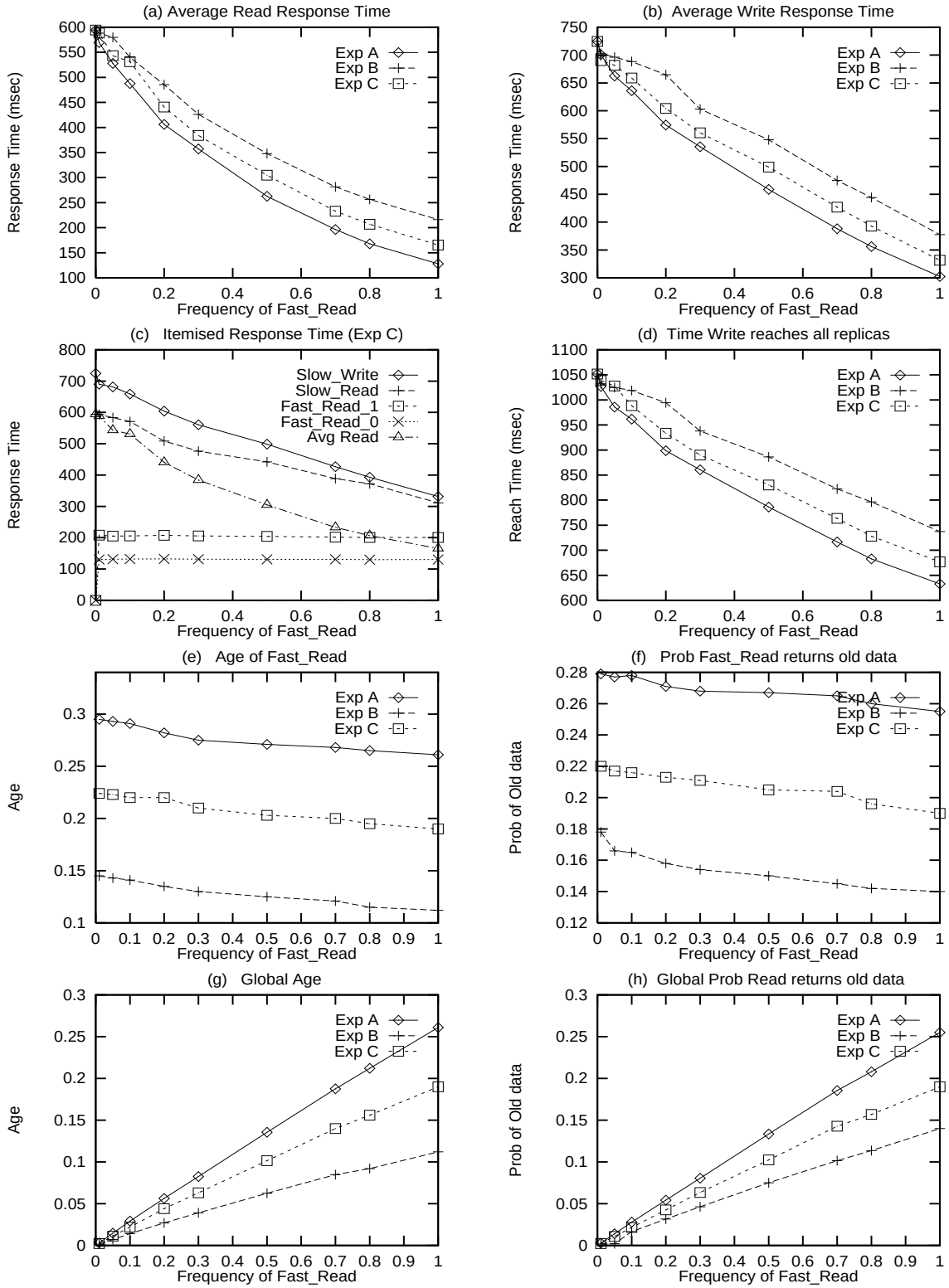


Figure 3: Varying read mix, writes are Slow_Writes

The time at which updates reach all replicas has shown a significant improvement (Figure 3(d)), which is due to the even distribution of the load among the resources, the decrease in the commit time of updates and the reduction of the load on the communication links which increases the speed at which updates are propagated.

The penalty paid by converting Slow_Read to Fast_Read is that fast reads return old information. It is observed that for a point of $slow_r$=0.5; the probability of Fast_Read returning old information is quite low (around 27% (Exp A), 15% (Exp B) and 20% (Exp C)), and the age of information returned by Fast_Read is around .27 (Exp A), .12 (Exp B), and .2 (Exp C) as shown in Figure 3(e) and (f). Further, the global age is only 0.135 (Exp A), 0.062 (Exp B) and 0.1 (Exp C) while the global probability of returning old information is 13.5% (Exp A), 7.5% (Exp B) and 10.2% (Exp C) for $slow_r$ =0.5 as shown in Figures 3 (g) and (h). So, it is noticed that it is only a fraction of an update that it is missed (Age=0.2 could be interpreted as one update is missed once in five reads). As $fast_r$ increases, the age decreases: since the time at which updates reach all replicas decreases, then replicas are updated more quickly and hence less stale.

From Figure 3(e), it is observed that Fast_Read_1 offers better age than Fast_Read_0 with an improvement ranging between 51% to 57%. On the other hand Fast_Read_0 has better performance in terms of response time, nearly 41% lower than Fast_Read_1. Therefore, an even mix of $fast_{r0}$ and $fast_{r1}$ appears to be a good compromise that combines the two benefits.

As shown above, converting Slow_Read to Fast_Read improves all performance metrics with a little loss in age. So, the choice for the mix of $slow_r$ versus $fast_r$ depends on the application: to the extent that the application is ready to accept some loss in the age, as the performance gets better. It appears that having $fast_r$ ranging from 0.5 to 0.8 is a reasonable range that combines both benefits.

#### 4.1.2 Varying *pr_commit* when updates are Opt_Write

This experiment explores the effect of varying *pr_commit* when all writes are Opt_Write and compares it with the case when all writes are Slow_Write. Parameters are set as follows:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | *pr_commit* |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0.25 | 0.25 | 0.5 | 0.4-1 |

Results are plotted in Figure 4. At *pr_commit*=1, there is a significant improvement in the write response time (60.2%) when updates are Opt_Writes rather than Slow_Writes as shown in Figure 4 (a). This is due to that response returns to the user before commit takes place. The response time of Opt_Writes decreases as *pr_commit* decreases, however, this is due to: aborting writes reduces the load on the resources as shown in Figure 4 (c). The response time of reads associated with Opt_Write is the same as that of reads with Slow_Write when *pr_commit*=1. It decreases as *pr_commit* decreases, but again this is due to the reduction of the load on the resources due to aborting writes.

Figure 4 (b) shows the coverage time with *pr_commit* 1, 0.9 and 0.8. In comparison with the coverage of Slow_Write, the coverage time is constantly lower. For the first few nodes, it is much better than that of Slow_Write. The improvement is around 54.8% for the first node, 21% for the second node, 8% for the third node for *pr_commit*=1. Then, it follows the same curve as Slow_Write with an improvement of nearly 4%. This improvement is due to that optimistic nodes perform the updates before they commit, and hence are updated much faster which improves

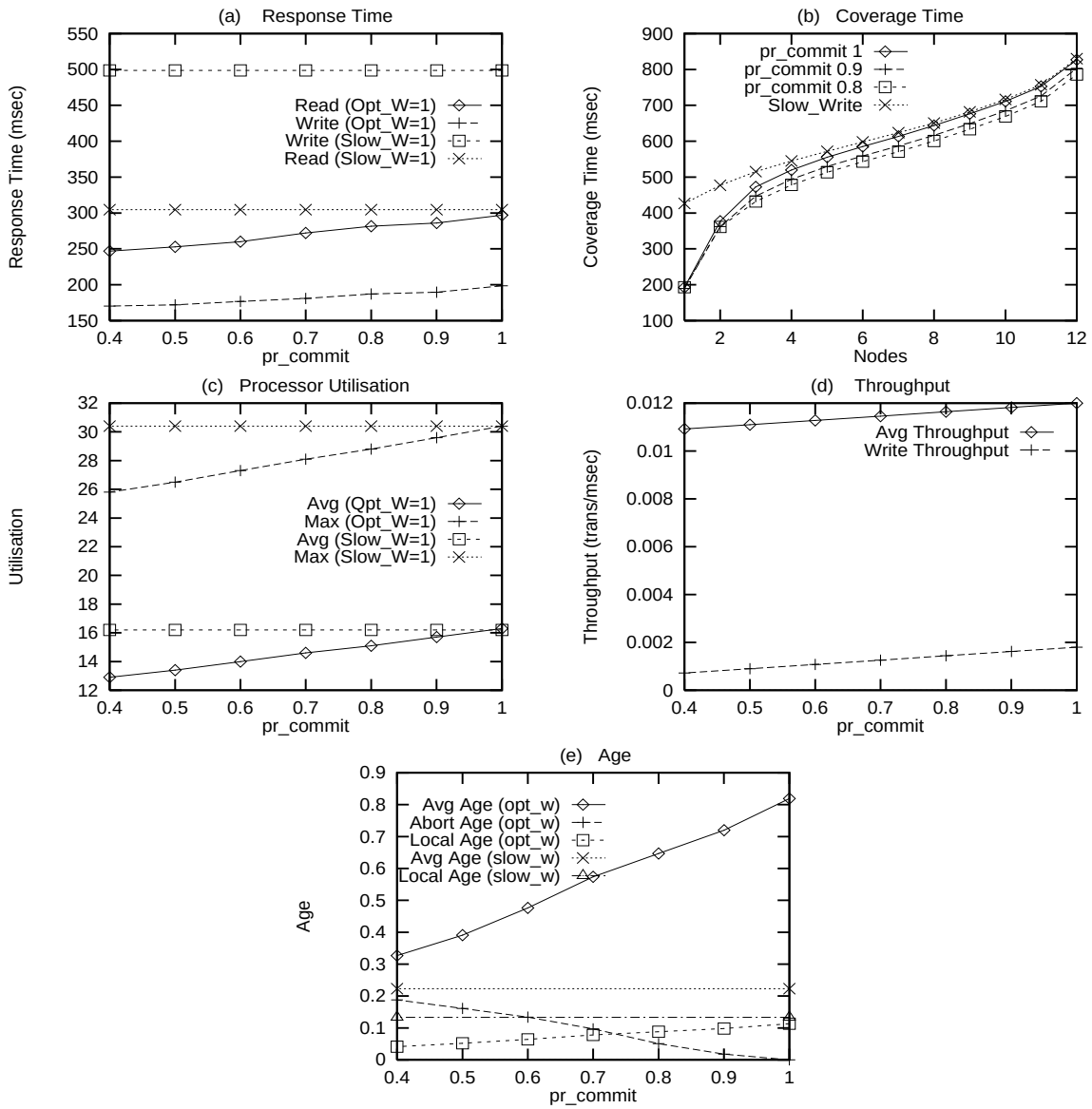


Figure 4: Varying *pr_commit*

the coverage. In the experiments, optimistic nodes are either 1 or 2 nodes. More improvements are expected when the number of optimistic nodes increases.

Obviously, the throughput decreases as the *pr_commit* decreases. As shown in Figure 4 (d), the decrease is nearly linear. The overall throughput decreases by 9% and the write throughput by 60% when *pr_commit* reaches 0.4.

The age returned by Fast_Read consists of the commit age, which is the age of information due to updates that commit, the abort age, which is the number of updates that aborts, and the average age is the sum of both. As shown in Figure 4 (e), the average age of Opt_Write decreases as *pr_commit* decreases. The commit age is dominating at *pr_commit*=1, then it decreases as *pr_commit* decreases since there are fewer updates that commit. On the other hand, the abort age increases and eventually dominates. In comparison with Slow_Write, the average age is higher with Opt_Write and this is due to that updates commit at local replicas and update the global version rather than updating it while assembling the quorum. So nodes at the other extreme of the tree experiences higher age. However, the local age is much lower than the average age and is even lower than the local age with Slow_Write. This is due to that updates are spread locally quicker, before they commit, as shown by the coverage time curve.

From there we can deduce that not all applications can benefit from Opt_Write. It is believed that applications that can guarantee that more than 80% of their Opt_Write will commit can combine the benefits of reduction in response time and coverage while not suffering from severe reduction in throughput. Further, it can still benefit from a low local age and does not suffer from an increase in the abort age.

#### 4.1.3 Varying Opt_Write versus Slow_Write

Opt_Write is expected to give better response time than Slow_Write, however, it applies the update to the replica before commit takes place. In this experiment, $slow_w$ versus $opt_w$ is varied to study the tradeoffs between Slow_Write and Opt_Write and to evaluate the performance of a combination of load mix between them. Parameters setting is as shown below; the remaining parameters are at their default values.

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ |
|---|---|---|---|---|---|
| 0-1 | 0-1 | 0 | 0.25 | 0.25 | 0.5 |

As shown in Figure 5(a), the average write response time decreases as $opt_w$ increases. This is because the response returns to the user before the update actually commits. As shown, the response time of Slow_Write, Opt_Write and read are basically the same; since converting Slow_Write to Opt_Write does not alter either the load on processors or the load on the links or the level of the data contention. Therefore, the improvement in write response time is nearly a linear combination of the response of Opt_Write and Slow_Write. As $opt_w$ increases, the coverage time decreases (Figure 5(b)), especially for the first few nodes (optimistic nodes), then it follows the same curve as Slow_Write.

However, as shown in Figure 5(c), the average age increases as $opt_w$ increases. The reason of the degradation is that updates commit at local replicas and update the global version rather than updating it while assembling the quorum. So nodes at the other extreme of the tree experience higher age. The local age is slightly better with Opt_Write due to the improvement in the coverage time. The abort age increases as $opt_w$ increases but it is quite low.

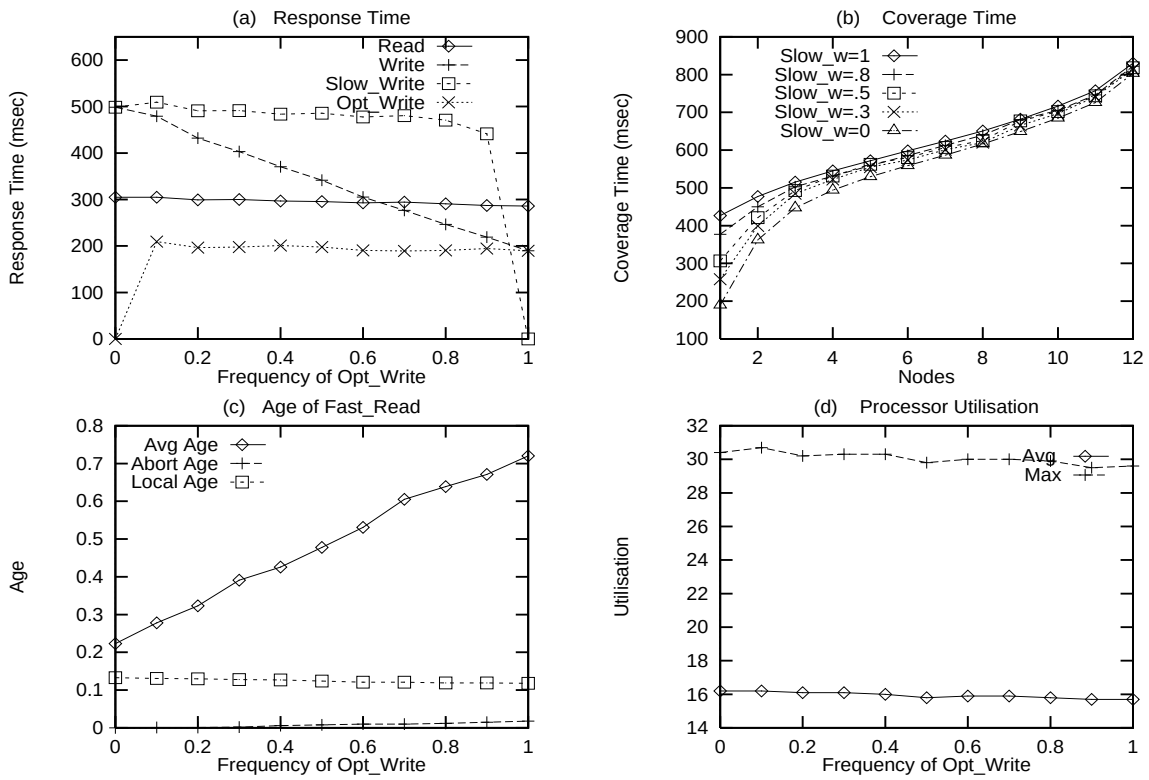


Figure 5: Varying $opt_w$ versus $slow_w$

Therefore, it seems that an even mix between Opt_Write and Slow_Write can compromise the benefit of reduction in write response time, coverage and local age with the loss in the overall age.

### 4.1.4 Varying the arrival rate

In this experiment, the arrival rate is varied to see how sensitive performance is to the load intensity. Two loads are considered where load A contains more synchronous read operations than the other load B. The parameter setting is as follows:

| Exp | $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | $inter_arr_rate$ |
|---|---|---|---|---|---|---|---|
| A | 0.5 | 0.5 | 0 | 0.25 | 0.25 | 0.5 | 600-1400 msec |
| B | 0.5 | 0.5 | 0 | 0.4 | 0.4 | 0.2 | 600-1400 msec |

For load A, as shown in Figure 6(a), the response time increases following an exponential curve. Steady state was not able to be reached for $inter_arr_rate$ less than 700 msec; limiting the overall throughput to 0.017 request/msec. It was observed that the responses of Slow_Write and Slow_Read are the ones dominating the performance (Figure 6(b)). Further, although utilisation of resources increases by increasing the arrival rates; they were still very lightly loaded as shown in Figure 6(f). It was observed that the commit time of Slow_Write follows exactly the same

curve and very close to the response of Slow_Write which means that it is the dominating factor (Figure 6(c)). The same holds for Slow_Read. Therefore, the bottleneck of the system lies in data contention (locks) not resource contention. The time at which updates reach all replicas also follows an exponential curve (Figure 6(d)); which is due the fact that the updates are propagated only after commitment, and are affected by contention for locks. Consequently, the age of information returned by fast reads as well as the overall age seen by all reads increase as shown in Figure 6(e).

Load B shows the effect of reducing $slow_r$ and increasing $fast_r$. It was observed that reducing $slow_r$ from 0.5 to 0.2 reduces the data contention and consequently, the response times of reads and writes reduce as shown in Figure 6(a), due to the decrease in the commit time. The reduction in read (write) is around 31% (19.5%) for *inter_arr_rate* 1000, it increases to 45% (23.4%) for *inter_arr_rate* 900 and continues increasing as the load increases. Further, it was able to cope with a higher arrival rate increasing the throughput by 16.8%. The time at which updates reach all replicas is better in B than A and the improvement is more obvious for high load: as updates in B commit earlier than in A and therefore spread faster. The resource utilisation is more balanced and the maximum load on the resources decreases while achieving the same throughput. This is because reducing $slow_r$ puts less load on nodes in the top cluster and the load is redirected to local nodes. The age returned by Fast_Read is nearly the same for low loads at A and B, but it is much less for B as the load increases as shown. This is because updates are propagated to other replicas faster than in A, hence the age returned is less. Consequently, this affects the overall age: it was interesting to see that although load B has only 20% of its read Slow_Read while A has 50%, the reduction in the age returned by Fast_Read has affected the overall age such that the overall age of B is only slightly higher than A for low loads and is much better than A for high loads.

From this one can deduce that increasing a load involving $slow_w$ and $slow_r$ slows down the system operation and affects the performance tremendously. Response time deteriorates as well as age while resources are underutilised and throughput is limited due to data contention. Therefore, this calls for applications that need better performance and throughput to relax consistency to reduce data contention; if they can afford it.

#### 4.1.5 Varying the mix between reads and writes

In this experiment, the mix between reads and writes is varied to explore the limitations that synchronous updates impose on the performance. Parameters setting is as follows:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | $freq_r$ |
|---|---|---|---|---|---|---|
| 0.5 | 0.5 | 0 | 0.25 | 0.25 | 0.5 | 0.65-0.95 |

Results are plotted in Figure 7. It is noticed that performance degrades as the frequency of writes increases. The resource utilisation increases but this is not the bottleneck of the system. Again data contention (locks) is the system bottleneck. This is because writes cause more data contention than reads as writes do not share locks while reads do. A steady state could not be reached for a frequency of writes more than 27%, when locks are saturated; limiting utilization of the resources to 23% on the average, and 36% maximum and write throughput to 0.0032 request/msec.

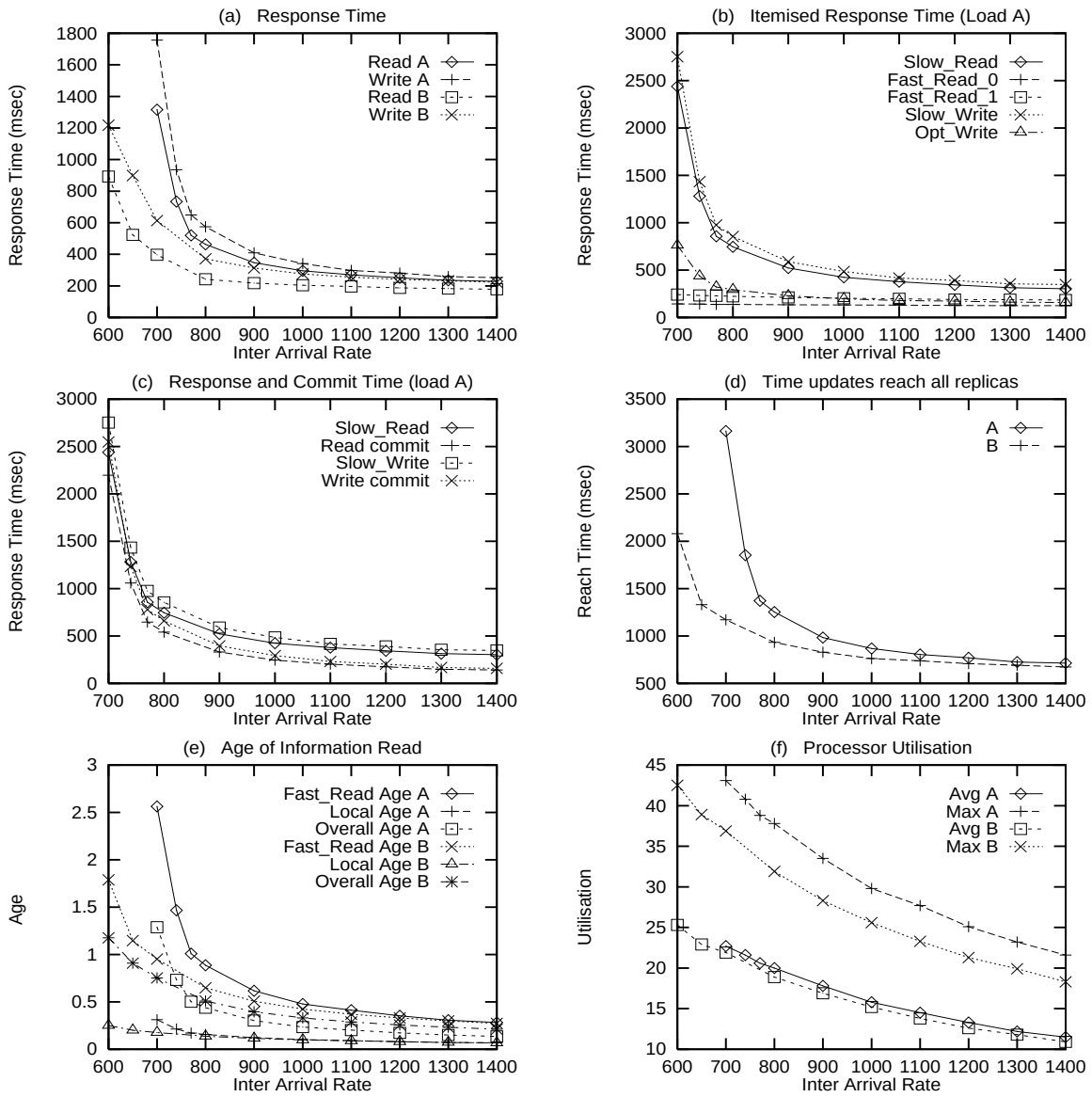


Figure 6: Varying the arrival rate

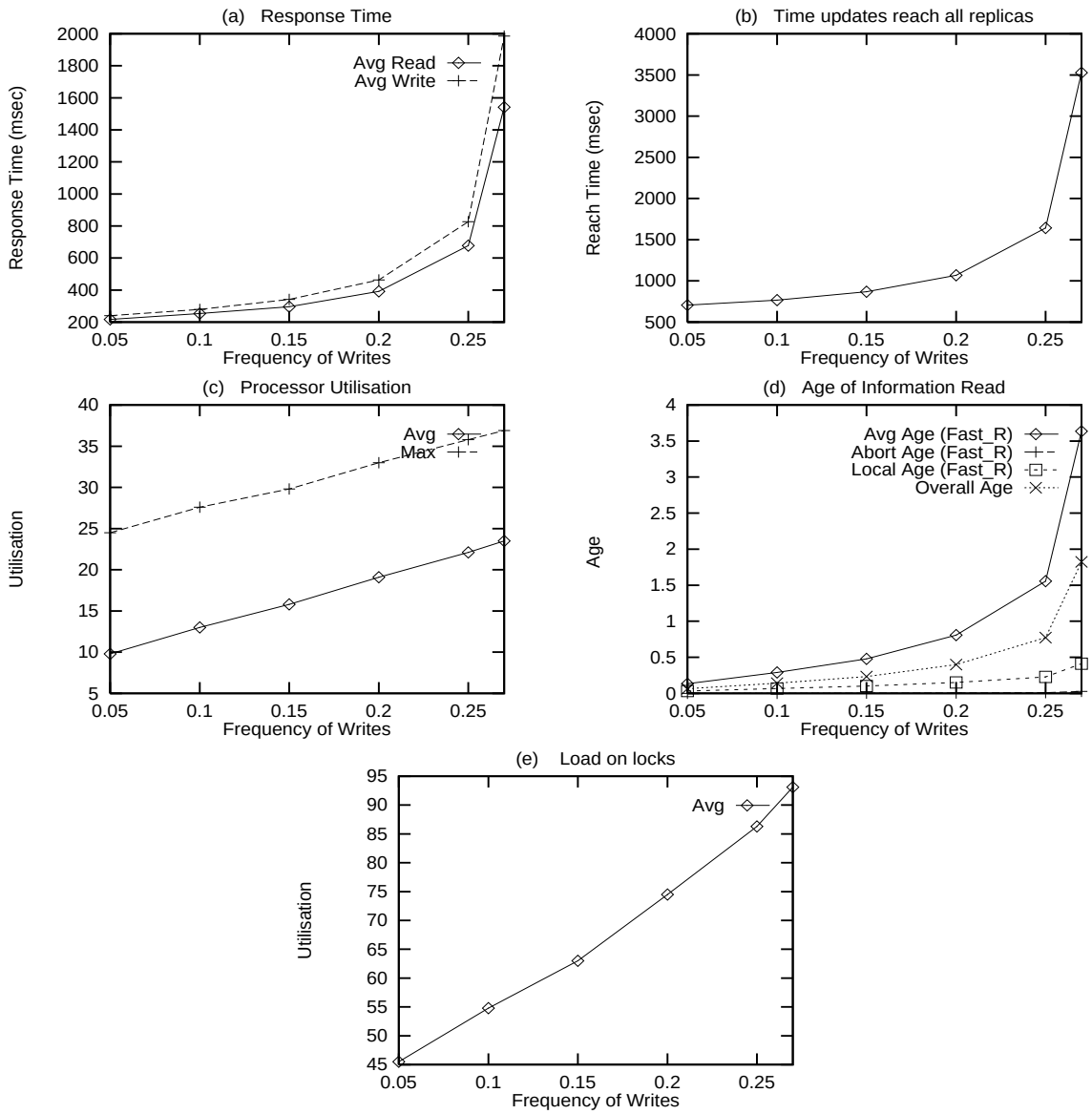


Figure 7: Varying the frequency of reads versus writes

#### 4.1.6 Varying the communication processing overhead

In this experiment, we explore the sensitivity of the performance to the communication processing overhead (*cpu_msg*). Results are shown for two load mixes, where load A contains more synchronous reads than load B. Parameters are set as follows:

| Exp | $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | *cpu_msg* |
|---|---|---|---|---|---|---|---|
| A | 0.5 | 0.5 | 0 | 0.25 | 0.25 | 0.5 | 0-20 msec |
| B | 0.5 | 0.5 | 0 | 0.4 | 0.4 | 0.2 | 0-20 msec |

As shown in Figure 8(a), the average response time of reads and writes increases slowly as *cpu_msg* increases then it follows an exponential curve. It was observed that the response time of Fast_Read_0 does not experience a noticeable increase while Fast_Read_1 increases slowly as *cpu_msg* increases since it involves communication (Figure 8(b)). Opt_Write increases slowly : mainly because nodes in the top cluster, their Opt_Write is effectively Slow_Write. As for Slow_Write and Slow_Read their responses increase sharply and are the dominating factors in the results. The reason for the increases is that CPU utilisation increases as *cpu_msg* increases and the increase is mainly due to communication processing (Figure 8(f)); this creates more contention on the CPU which obviously affects every operation. Consequently, synchronous operations take longer to commit since 2PC is involved with a fair amount of communication. The time each operation takes until it releases the locks increases and hence data contention increases. The commit time of reads and writes follows the same curve as their average response and is quite close which means that they are the dominating factors (Figure 8(c)). In brief, we can say that two interacting factors are affecting results:
1) the increase of CPU contention due to communication processing, and
2) the increase of data contention because requests spend longer time in the system

The time at which updates reach all replicas and the age returned by fast reads (Figures 8(d) and (e)) increase as *cpu_msg* increases since updates take longer to commit and hence propagate to other replicas slowly.

Comparing load A with B, it is obvious that load A is more sensitive to *cpu_msg* than load B. As shown in Figure 8(a), the response time of reads and writes increases more sharply with load A than load B and reaches saturation more quickly. For instance for *cpu_msg*= 5,8 and 12 msec, there is an increase (w.r.t. the ideal case of *cpu_msg* =0) of 30.7%, 67.1% and 148% for reads with load A, as opposed to 16.3%, 30.5% and 53.5% with load B. For writes the increase is of 28.7% , 67.3% and 152% with load A as opposed to 19%, 42% and 72.5% with load B. It was observed that the load that A places on the processors for communication is high (35% for *cpu_msg* =15 for nodes in the top cluster) and it is higher than the load that B places. Consequently, the locks are saturated more quickly. Again the data contention is the main reason behind the increase in the response time, which consequently increases the reach time and age as shown. The overall age of load A is slightly lower than that of load B for low values of *cpu_msg*. However, as *cpu_msg* increases, the age of Fast_Read of A increases and the overall age bypasses that of B.

From this experiment, we can deduce that performance is quite sensitive to high values of *cpu_msg* due to high CPU contention and data contention, specially when the load mix has high frequency of synchronous operations.

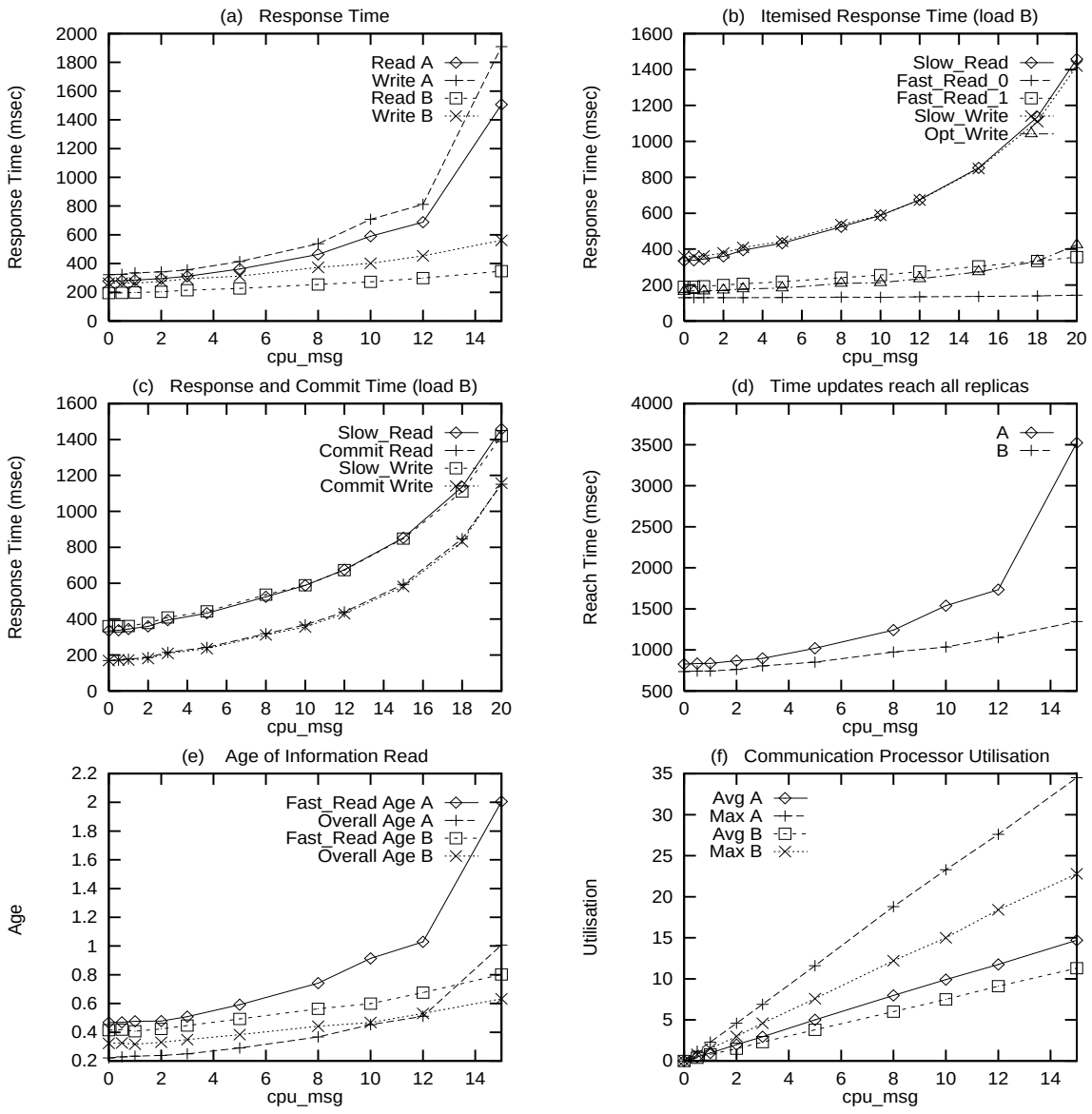


Figure 8: Varying the communication processing overhead

### 4.1.7 Varying the network delay

The following experiments explore the sensitivity of performance to the delay on the network. The ratio of *lan_del:wan_del* is varied from 1:10 to 1:5000. First, *wan_del* is fixed to 10 msec and *lan_del* is varied from 0.002 to 1.0 msec. Then, *lan_del* is fixed to 0.02 msec and *wan_del* is varied from 0.2 to 100 msec.

Results are shown for two load mixes; where load A contains more synchronous reads than load B. Parameters setting is as follows:

| Exp | $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ |
|---|---|---|---|---|---|---|
| A | 0.5 | 0.5 | 0 | 0.25 | 0.25 | 0.5 |
| B | 0.5 | 0.5 | 0 | 0.4 | 0.4 | 0.2 |

The performance has shown to be is insensitive to varying *lan_del*, as shown in Figure 9. This is expected since it has low values so it does not have a great impact on performance in comparison to *wan_del*. Therefore, this parameter can be factored out.

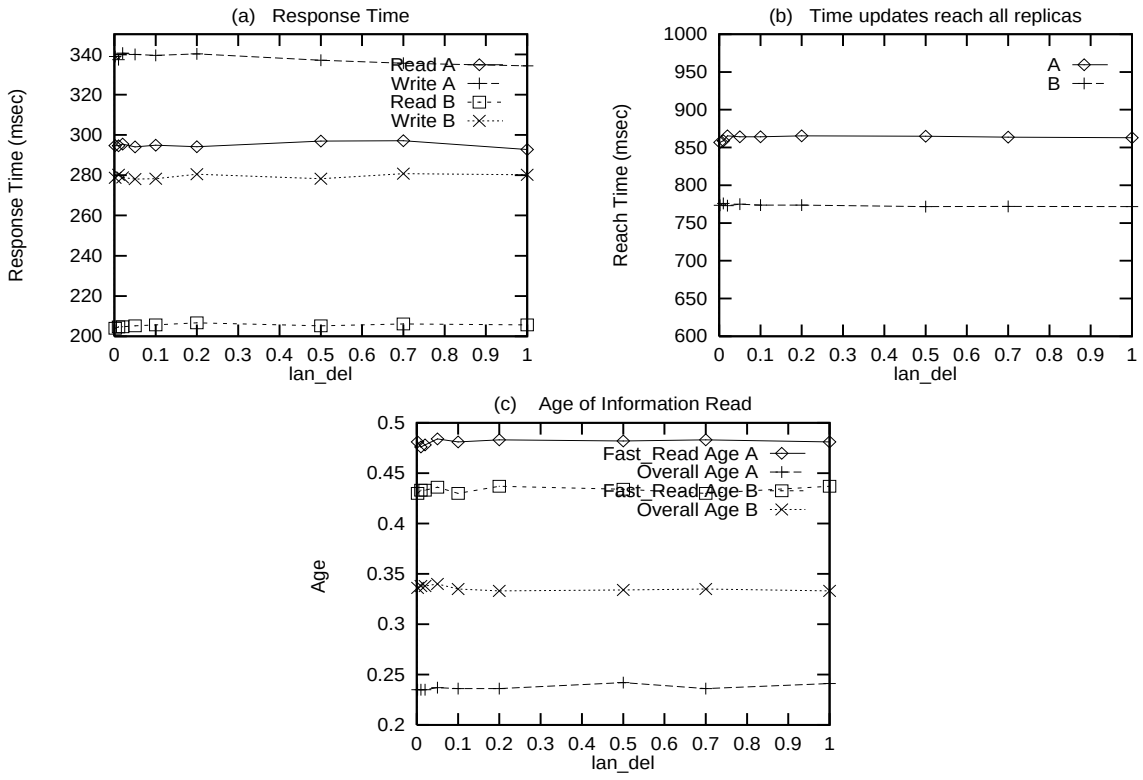


Figure 9: Varying the network delay (*lan_del*)

As shown in Figure 10, the response times of reads and writes are very sensitive to *wan_del*. They are quite low for values of *wan_del* less than 20 msec. The increase in the response time of reads is 25% (43%) and writes is 31% (46%) for load B (A) compared to the ideal case when *wan_del*=0. Then, they increase sharply. It is obvious that load A is more sensitive to

*wan_del* than load B and saturation is reached more quickly. It was observed that the response of Fast_Read_0 is not affected at all (Figure 10 (b),(c)). The response of Fast_Read_1 increases slowly following nearly a linear curve. The response of Opt_Write increases slowly till a point (50 msec load B, 30 msec load A) when the curve turns to be exponential; because nodes in the top cluster their Opt_Write are effectively Slow_Write. The responses of Slow_Write and Slow_Read increase sharply for both A and B and dominate the results. Again, the resources were found to be lightly loaded and even underutilised, and not the source of any major contention. However, locks were highly contended and are the bottleneck of the system. The reason is that: while *wan_del* increases, the time it takes a request to assemble a quorum and commit increases and it holds locks for a longer time. Consequently, the time other requests wait for locks increases. Figure 10(e), plots the age returned by Fast_Read with loads A and B as well as the overall age: the age returned by Fast_Read with A is higher than B since updates take longer to commit and consequently take longer for reaching other replicas. The overall age of A is slightly lower than B (0.1 updates lower) for low values of *wan_del* then the gap decreases and eventually it becomes higher than B for high values of *wan_del*.

From here it can be concluded that performance is very sensitive to *wan_del*. Synchronous operations suffer and dominate the degradation in performance with the increase of the delay. This suggests that if applications can accept observing slightly stale information then they should increase their frequency of Fast_Read especially when the network delay becomes higher as this offers better performance as well as lower age.

### 4.2 Varying network topologies when updates are synchronous

In the following experiments, we evaluate the performance of the protocol for various physical (and hence logical) networks, when updates are synchronous. Since the number of possible topologies can be infinite, we have limited our choices to a 12-nodes network, three of them are fully connected through long distance links to form a backbone and the remaining nodes are grouped into 3-clusters each forming a LAN. The connections between the clusters are varied to reflect certain aspects that might affect performance. Sections 4.2.1 through 4.2.4 consider hierarchical networks while Section 4.2.5 considers a non-hierarchical network.

The different hierarchical networks considered are shown in Figure 11. As shown, Net 2, Net 3 and Net 4 explore the effect of increasing the number of levels of the hierarchy. Net 2 spans three levels by connecting one of the LAN clusters to the backbone, and the two other clusters are linked to the backbone through that cluster. Net 3 is organised like a chain, spanning four levels. Net 4 shows the effect of connecting the two clusters in the bottom level -forming one logical cluster- and both clusters are connected through the same node (node 5) to the higher level. Net 5 explores the effect of connecting the three LAN-clusters at the bottom levels, forming one logical cluster, which is linked to the backbone via node 1.

For this set of experiments, the read mix is varied and the results are compared with those of Net 1. The parameters setting is as follows:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ |
|---|---|---|---|---|---|
| 0.5 | 0.5 | 0 | 0-1* | 0-1* | 0-1 |

* The ratio between $fast_{r0}$ to $fast_{r1}$ is kept 1:1

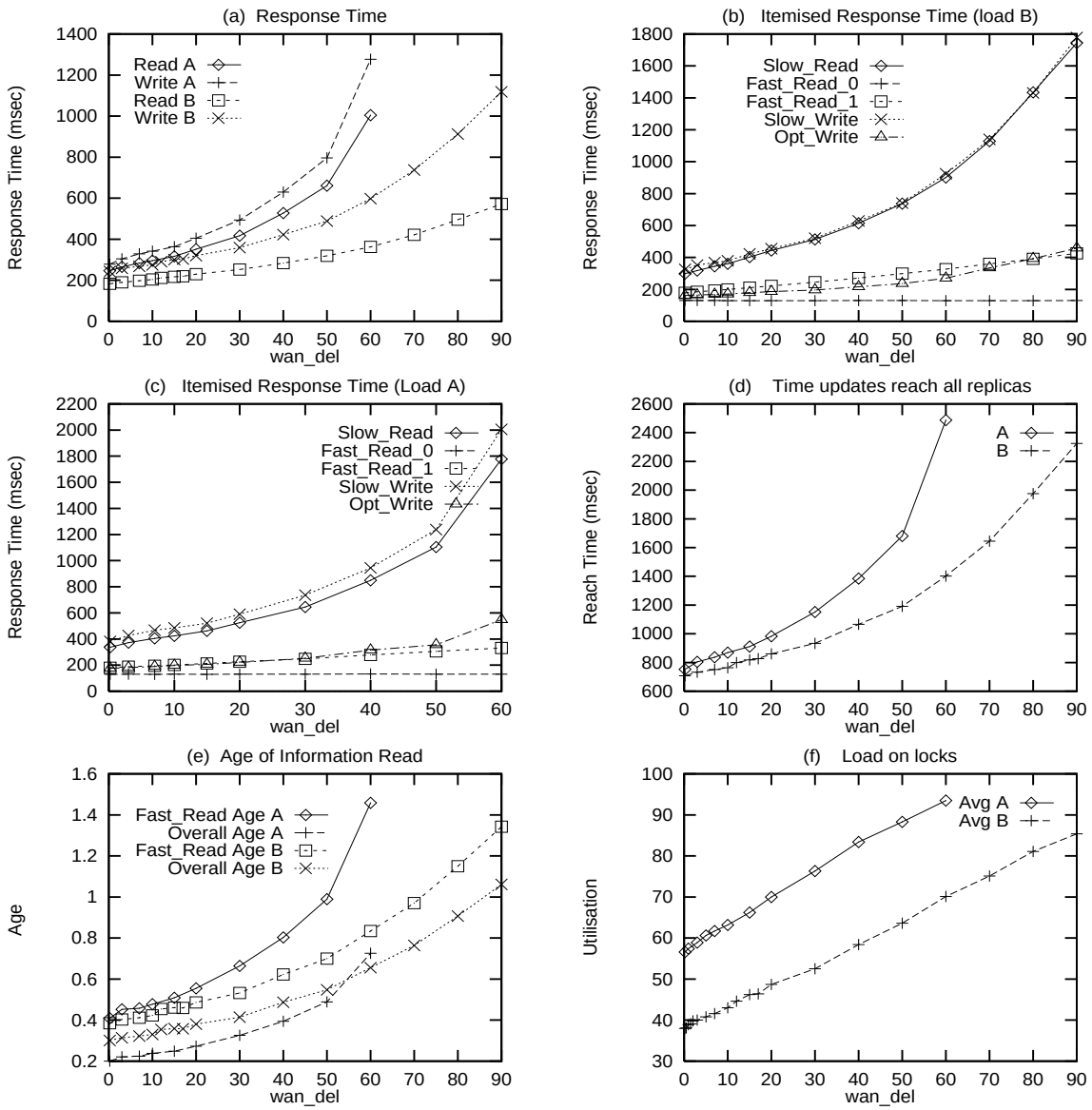


Figure 10: Varying the network delay (*wan_del*)

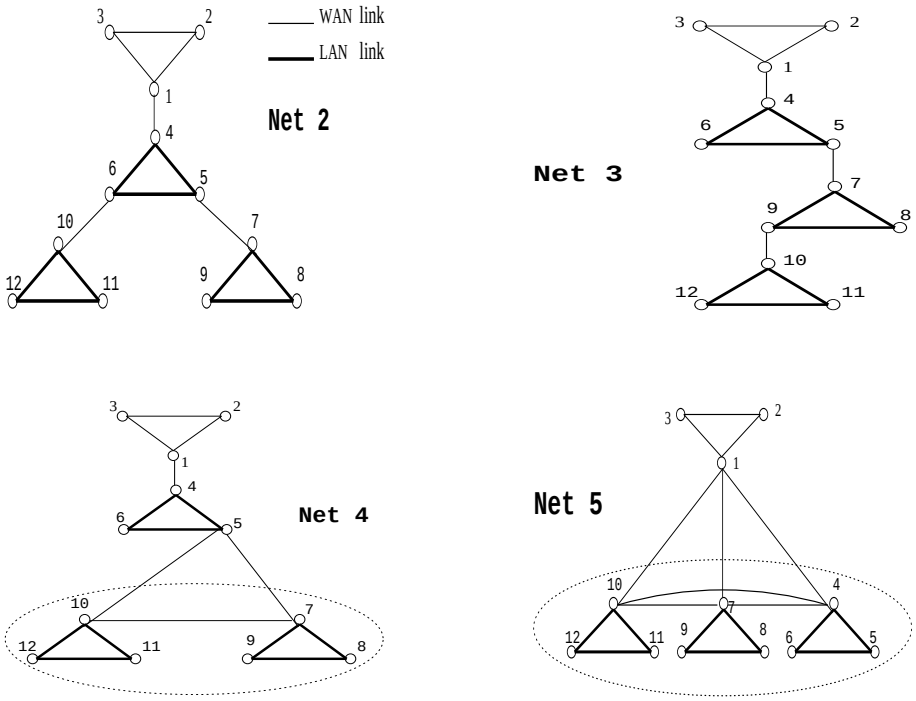


Figure 11: Different hierarchical network topologies for 12 replicas

### 4.2.1 Network Net 2

It is observed that the average write response time on Net 1 experiences lower response time than on Net 2 as shown in Figure 12(a). The difference is larger for higher frequency of Slow_Read then it reduces as more Fast_Reads are introduced. There are two reasons for the reduction in response:

1. on Net 2 there is an extra load placed on node 1 which acts as a CO for most of the nodes, hence this incurs an extra processing overhead (communication and 2PC) as well as data contention. Therefore, the time it takes Slow_Write to commit on Net 2 is higher than Net 1 as shown in Figure 12(b). It is observed that the difference in commit time is larger with higher frequency of Slow_Read then it decreases as less contention is present.

2. since Net 2 consists of 3 levels while Net 1 consists of only 2; Slow_write take more time from leaf nodes to send requests and receive results from CO. Further, the traffic on link between nodes 1 and 4 is higher than in Net 1. This increases the communication time spent in both processing and transmission. Hence, the response time increases.

The response time of Slow_Write (Figure 12(b)) is observed to be constantly lower for Net 1 than Net 2. The difference is large for high frequency of Slow_Read as both factors above affects the results. Then, it reduces as mainly the second factor only affects results. Since Opt_write is approximately the same for both networks and since the response of Slow_Write is the one

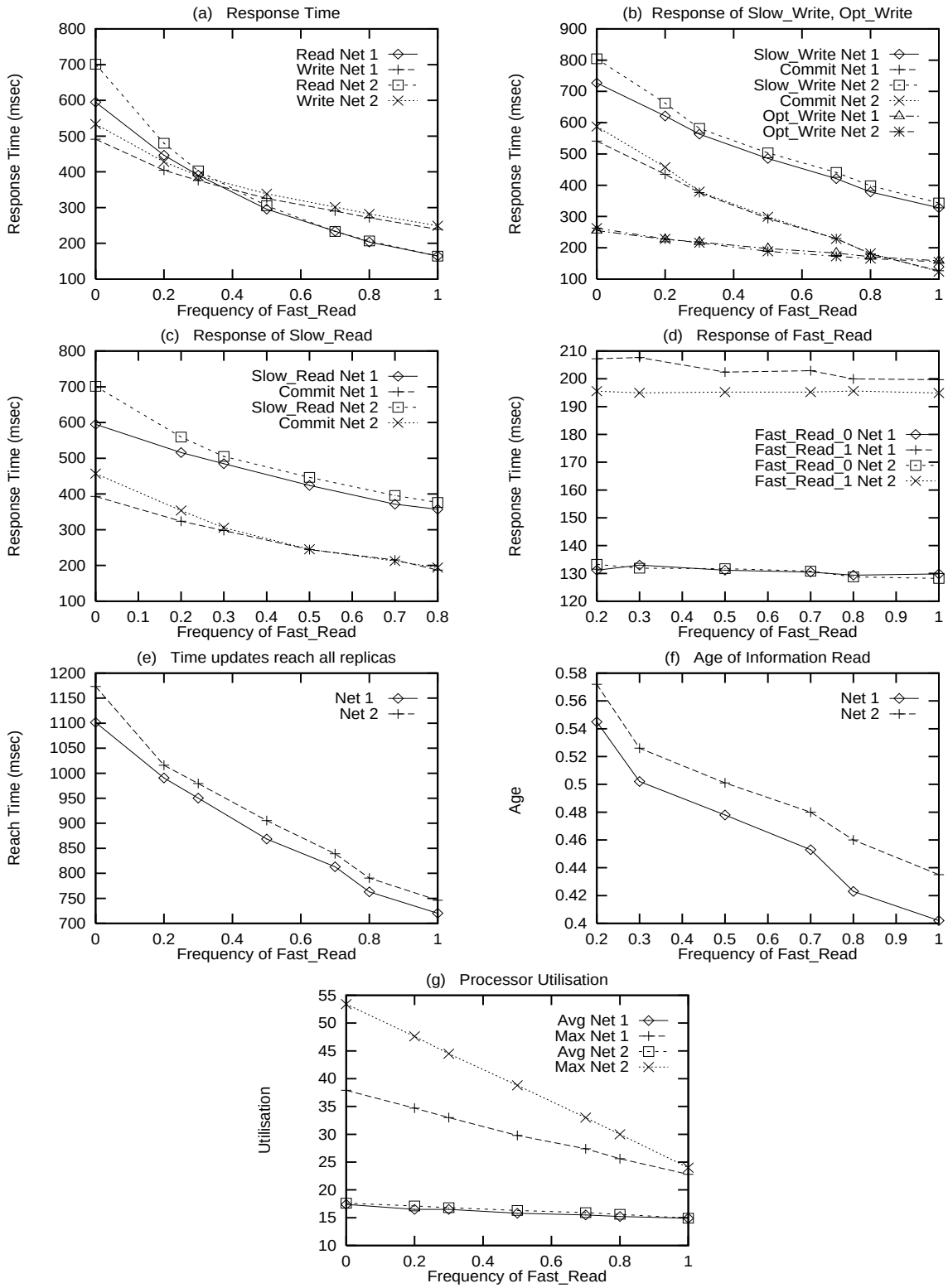


Figure 12: Network 2

dominating the results, then the comments on the response of Slow_Write apply to the average write response time (Figure 12(a)).

For the response of Slow_Read (Figure 12(c)), the same comments apply as for Slow_Write. The response of Fast_Read_0 and Fast_Read_1 are plotted in Figure 12(d). The response of Fast_Read_0 is more or less the same for both networks. Fast_Read_1 experiences lower response time on Net 2 than on Net 1 especially for high frequency of Slow_Read. The reason is that for Net 2, half of the nodes grant their requests from parents which are not in the top cluster, hence less loaded, while in Net 1 all nodes grant their requests from the top cluster. However, as $fast_r$ increases, the load on the top cluster of Net 1 reduces; due to less 2PC and communication overhead as well as lower load of executing Slow_Read requests. Consequently, the performance of Fast_Read improves on Net 1 and is more or less the same on both networks. Therefore, the average performance of reads for Net 2 (Figure 12(a)) is higher than Net 1 for low values of $fast_r$, when the response of Slow_Read is dominating, then it reduces to approach Net 1 and they are both approximately the same for high values of $fast_r$.

Since updates take longer on Net 2 to commit and since propagated updates have to travel down two levels instead of one; then the time updates reach all replicas in Net 2 is constantly higher than Net 1 as shown in Figure 12(e). Consequently, the age returned by Fast_Read is higher as shown in Figure 12(f).

From there we can conclude that: Slow_Read, Slow_Write and Opt_write perform better on Net 1 than on Net 2 as the load is evenly distributed among the nodes in the top cluster and the communication time is less between leaf nodes and CO's. The performance of Fast_Read_0 is the same while Fast_Read_1 has a better response time on Net 2, since not all requests are granted from the top cluster. However, the age returned by Fast_Read_0 and Fast_Read_1 are worse than Net 1. It should be noticed that these differences are expected to be higher with higher loads: as noticed the processors are quite underutilised, however, we were not able to take measurements for higher loads since data contention was quite high and steady state could not be reached.

#### 4.2.2 Network Net 3

As shown in Figure 13, the results of this experiment are nearly the same as with Net 2 and the same comments apply. We notice that the difference in response time of Slow_Write, Slow_Read as well as the average write and read operations are bigger with Net 3 than with Net 2. This is because Net 3 consists of 4 levels, hence transactions spend more time in communication in order to send requests and receive results from COs. The commit time is the same since the resource utilisations are more or less the same so is the level of data contention. The reach time is higher in Net 3 due to higher communication time and consequently the age is higher.

#### 4.2.3 Network Net 4

It is observed that the results were very similar (nearly identical) to those of Net 2 and they were not plotted. It was observed that there is very a slight degradation in performance over Net 2, due to the extra overhead placed on node 5 due to the extra communication overhead as well as the processing of Fast_Read_1 requests. The only noticeable degradation is for the response of Fast_Read_1 which increases with around 8-10 msec over Net 2 due to the extra overhead placed on node 5. This is mainly because half of the nodes grant their requests from node 5 which increases the contention on that node.

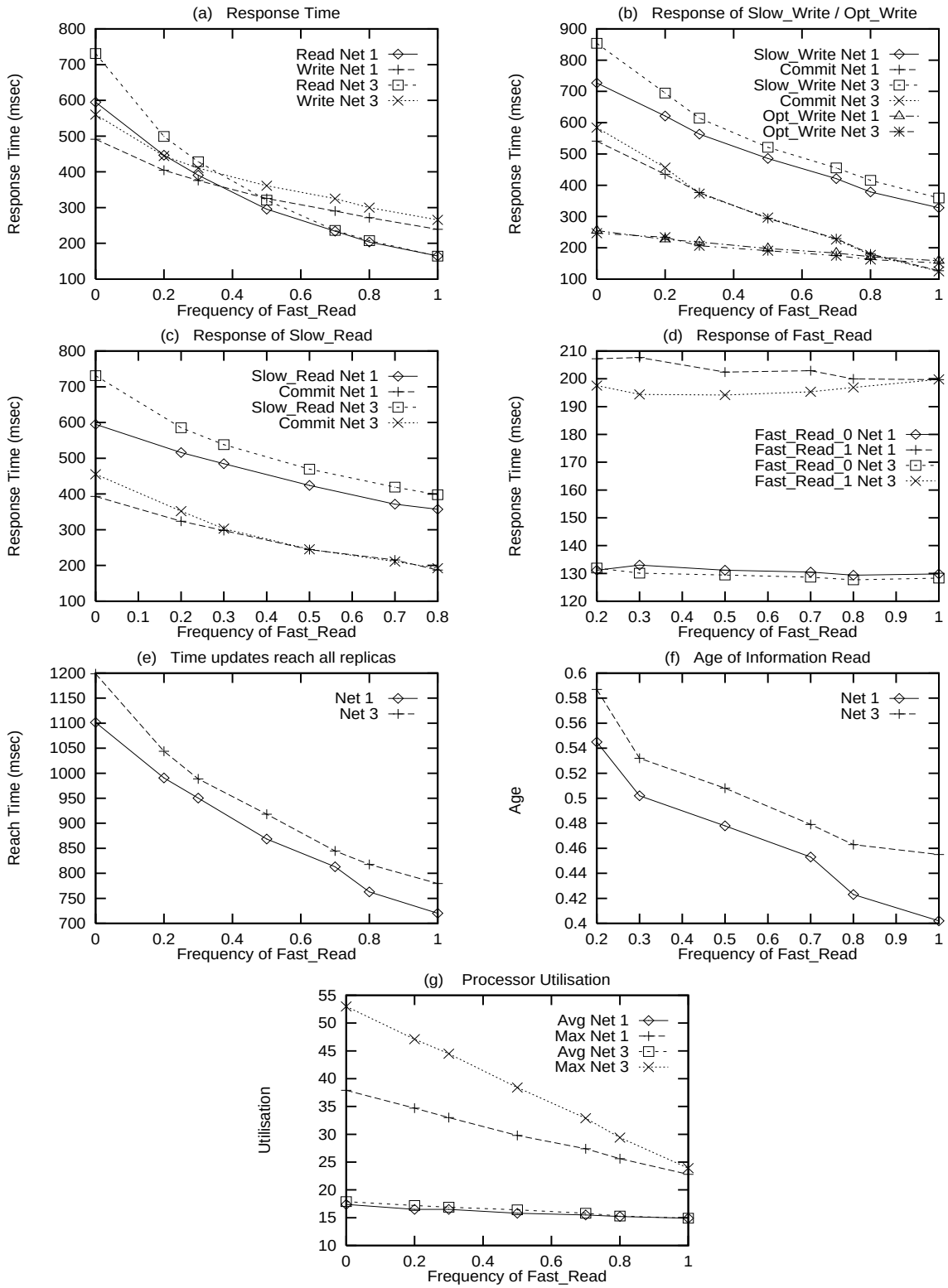


Figure 13: Network 3

#### 4.2.4 Network Net 5

On this network, performance is worse than on Net 1 for all read mixes and for all metrics. The reason is mainly due to the extra load Net 5 puts on node 1:

- with Slow_write, Slow_Read and Opt_Write operations, node 1 acts as CO for most of the nodes. Therefore, it suffers from more overhead: communication and 2PC.

- with Fast_Read_1: node 1 acts as the parent of most nodes, hence, it performs all their Fast_Read_1 requests. Therefore, it suffers from extra processing and communication overhead.

Therefore, node 1 is more loaded than on Net 1. This can be seen from Figure 14(g) where the average utilisation of processors is more or less the same for both networks while the maximum (which happens to be the load on node 1 in Net 5 and the load on nodes 1,2 and 3 in Net 1) is constantly higher than Net 1 by around 17%. For high frequency of Slow_Read, this increase is because of the 2PC and communication overhead. For high frequency of Fast_Read, this is because of executing Fast_Read_1 requests as well as communication overhead. This extra load causes the commit time for Slow_Write and Slow_Read to be higher on Net 5, which increases the data contention. Consequently, the response time increases, as shown in Figures 14(b) and (c). In Figure 14(d), it is observed that the response of Fast_Read_0 is not affected while that of Fast_Read_1 is constantly higher than Net 1 by around 28 msec due to the extra load on node 1. Consequently, as shown in Figure 14(a), the average response time of reads and writes are constantly higher than on Net 1.

For the reach time (Figure 14(e)), it is slightly higher than on Net 1, especially for high frequency of Slow_Read then it is approximately the same. Consequently, the age (Figure 14(f)) is also higher for high frequency of Slow_Read then reduces to approach that of Net 1. It should be noticed that on Net 5, an update on a leaf nodes needs to cross a maximum of 2 wan links to reach all other replicas, while on Net 1 it needs 3 hops. Hence, the communication time to propagate an update on Net 5 is less than Net 1. However, the increase in the commit time on Net 5 is higher than the decrease in the communication time and causes the reach time to be higher than in Net 1.

#### 4.2.5 Network Net 6

This experiment shows the performance over a network where all nodes are in one logical cluster, more or less fully connected (Net 6), as shown in Figure 15. Applying synchronous operations on such a network represents the performance of standard protocols used in the literature, more precisely, using majority quorum consensus over the whole network. Since there are no levels in the hierarchy, then Opt_Write is converted to Slow_Write and Fast_Read_1 is converted to Fast_Read_0. In order to compare results with those of Net 1, the experiment varies a read mix consisting of Slow_Read and Fast_Read_0 when all updates are Slow_Write. That is the parameters setting is as follows:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0-1 | 0 | 0-1 |

The results are plotted in Figure 16 and it has been observed that:

• Steady state was not able to be reached on Net 6 for $slow_r > 0.5$. Assembling a quorum from seven nodes (versus two in Net 1), increases data contention. Further, since each node can be a CO, the rate at which deadlock occurs is higher. Finally, resources are more loaded since

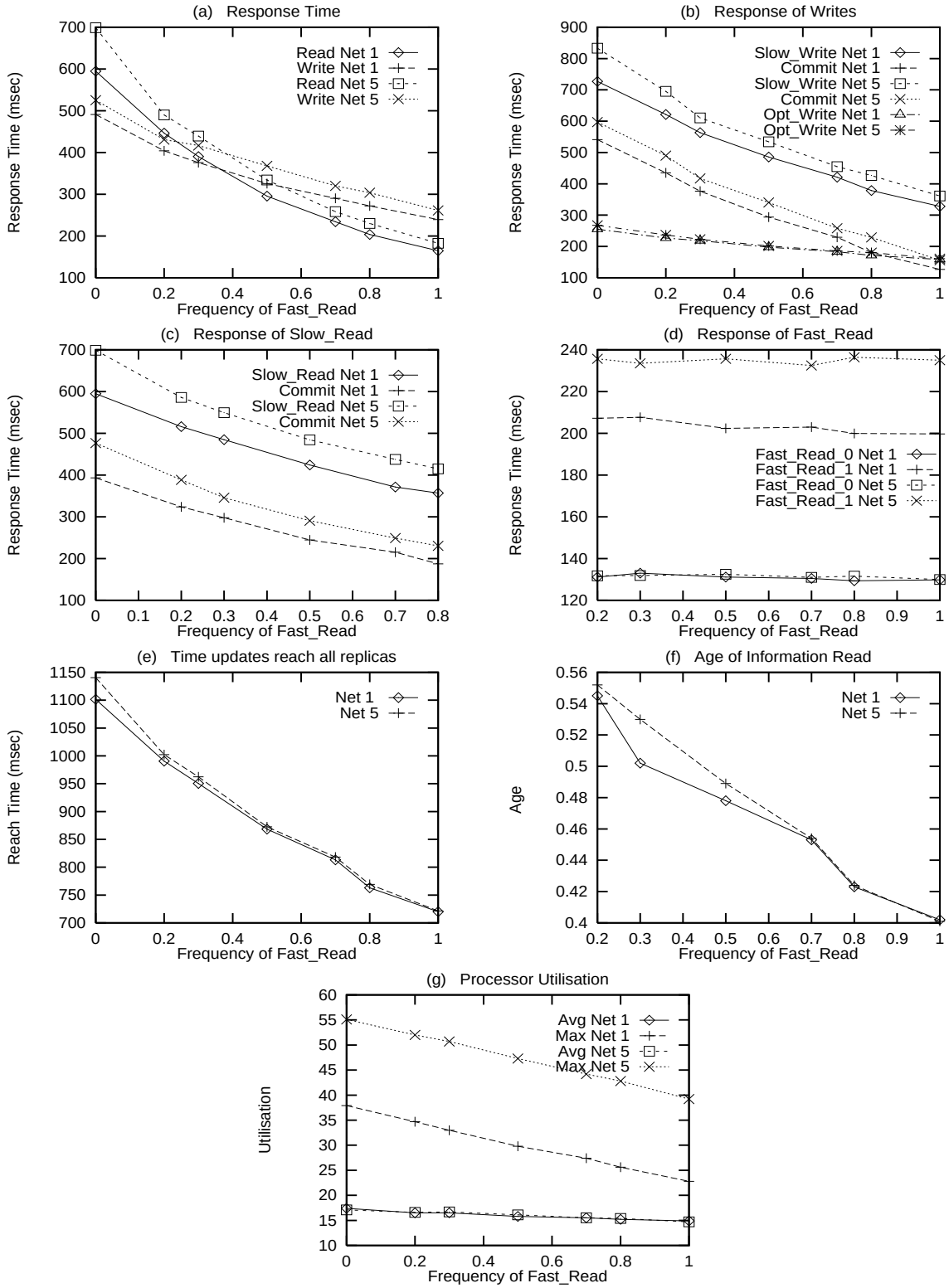


Figure 14: Network 5

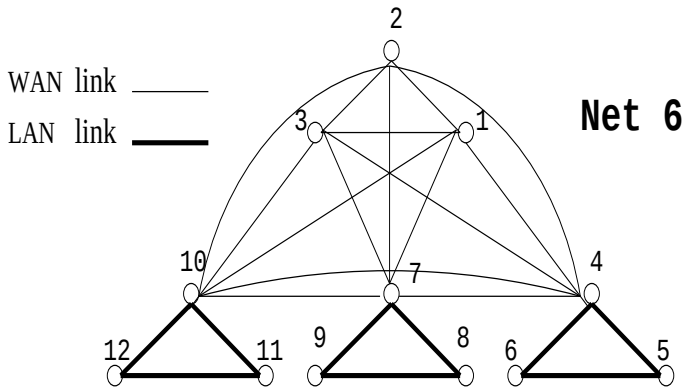


Figure 15: A non-hierarchical network, Net 6

all nodes are involved with 2PC and communication overhead. These factors increase the time by which operations commit and eventually no steady state could be reached.

• For loads where steady state could have been reached, most of the performance metrics showed a degradation. The average response time of writes is higher than in Net 1, with a degradation between 46% and 52%, as shown in Figure 16, due to high commit time. Although in Net 6 Slow_Write and Slow_Read avoid the communication time needed to send requests and receive results from CO, this does not compensate for the huge increase in the commit time. The average response time of reads on Net 6 is higher than Net 1 for high $slow_r$, then it decreases to approach Net 1 when all reads are Fast_Read_0.

The time at which updates reach replicas for Net 6 is higher than Net 1. Although Net 6 needs to cross fewer links than Net 1 to propagate updates to other replicas not participating in the quorum, the increase in the commit time is higher than the reduction in the communication time. The age returned by Fast_Read in Net 6 is lower than that of Net 1 for high frequency of Fast_Read, as shown in Figure 16. Since a majority of the nodes participate in the quorum, they are up-to-date, and since Fast_Read is performed directly from them, this explains the lower age. However, as $slow_r$ increases, the commit time of updates increases which slows down the speed of propagation. Consequently, the age in Net 6 increases and bypasses that of Net 1 when more than 30% of the reads are Slow_Read as shown.

In conclusion, it is observed that performance of synchronous operations on Net 6 is much worse than Net 1. Throughput is limited tremendously, especially with high loads. Within Net 1, HARP is able to achieve better throughput for the same types of operations since it assembles a smaller quorum. The response times of Slow_Write, Slow_Read and Fast_Read_0 are worse than Net 1, as well as the time at which updates reach all replicas. The gain in the age over Net 1 is only for high frequency of Fast_Read, and the improvement is of a maximum of 21%, which appears to be much less than the losses in all other metrics.

### 4.3 Experiments with Fast_Writes

In this section, several experiments are presented where updates are Fast_Write. A number of parameters are varied such as the read mix, the load intensity and the communication overhead

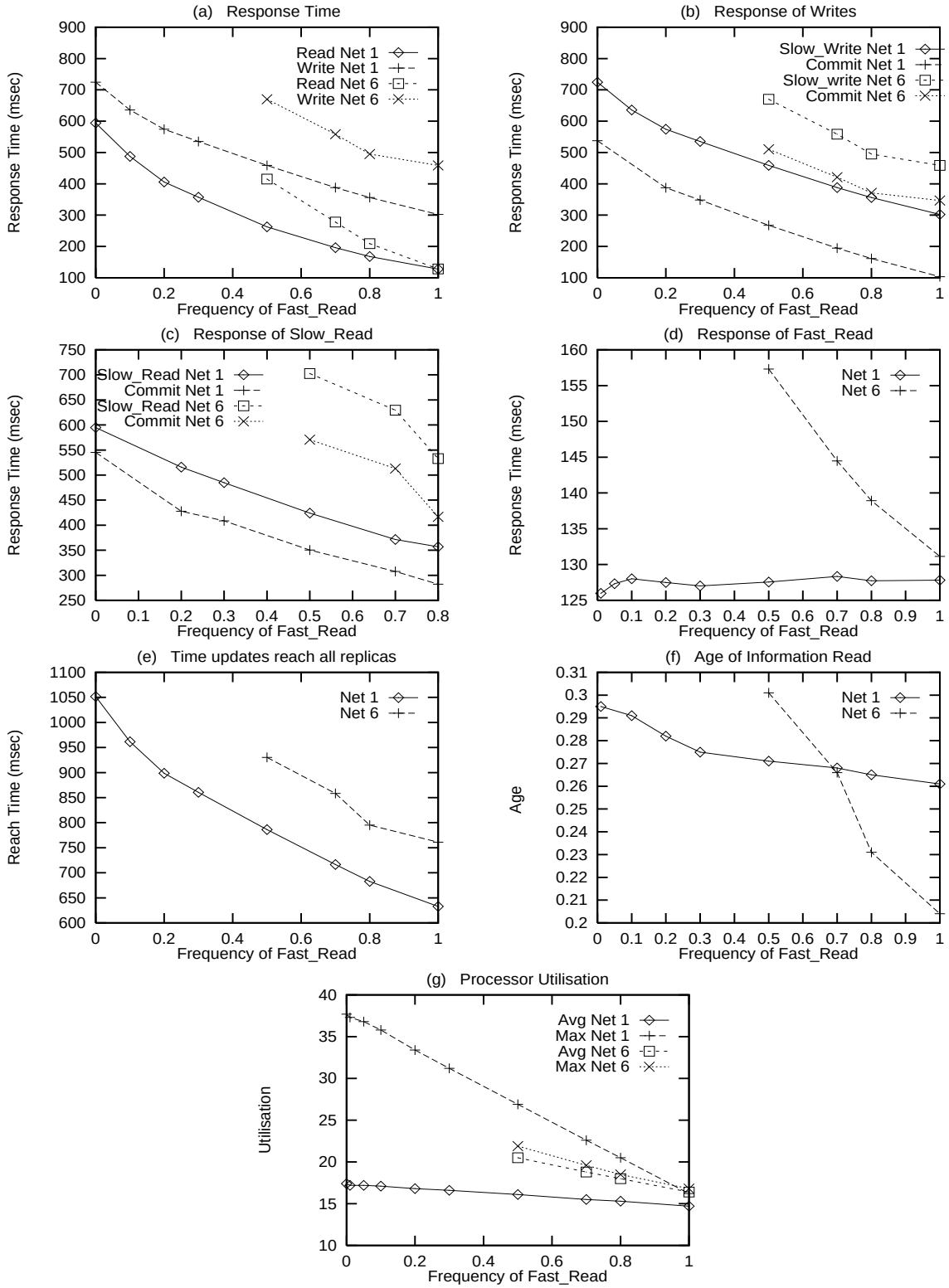


Figure 16: Network 6

to observe how sensitive asynchronous operations are to those parameters. Also, the results are compared with those when updates are synchronous. These experiments are performed over the physical network Net 1.

#### 4.3.1 Comparing Slow_Write versus Fast_Write with a mix of Fast_Reads

In this experiment, a mix of Fast_Read is varied for two loads: load A where all updates are Fast_Write and load B where all updates are Slow_Write. The parameters setting is as follows:

| Exp | $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | $freq_r$ |
|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0-1 | 0-1 | 0 | 0.7 |
| B | 1 | 0 | 0 | 0-1 | 0-1 | 0 | 0.7 |

The average response time of reads and writes are plotted in Figure 17(a). There is a huge difference between the response of Fast_Write and Slow_Write. Also, the increase in the response of Slow_Write with the increase of $fast_{r1}$ is much larger than that of Fast_Write (for $fast_{r0}$=1, the response of Slow_Write is 5 times greater than Fast_Write, for $fast_{r0}$=0, it is 8.5 times greater). The reason for the sharp increase is that as $fast_{r1}$ increases, higher load is placed on the resources, especially on those of the top cluster (Figure 17(c),(d)). Therefore, the commit time of Slow_Write increases as well as data contention. The average response time of reads with Slow_Write is slightly higher than with Fast_Write because resources with Slow_Writes are more loaded due to 2PC and communication overhead. This is more illustrated in Figure 17(b), where it is shown that both Fast_Read_0 and Fast_Read_1 experience higher response with Slow_Write than with Fast_Write.

The time at which updates reach all replicas, Figure 17(e), is much higher for Slow_Write than Fast_Write and it increases sharply as $fast_{r1}$ increases (it is 27% higher for $fast_{r1}$=0 and 95% higher for $fast_{r1}$=1). This is mainly due to the load Slow_Writes put on nodes of the top cluster which increases the commit time and consequently increases reach time. Also, the links are more loaded due to 2PC, communication and propagation overhead which increases the communication time. Similarly, the coverage time was observed to be much better with Fast_Write than Slow_Write as shown in Figure 17(f). The age returned by Fast_Reads is plotted in Figure 17(g). As shown, Fast_Read with Slow_Write have better age than those with Fast_Write. This is expected as replicas in the top cluster with Slow_Write are updated first, hence, the age, especially that of Fast_Read_1, is better. Further, with Fast_Write leaf nodes suffer from higher age because updates originating at other leaf nodes are committed locally, and the *global_version* is updated immediately, rather than after the update commits at the top cluster which increases the age. However, since the reach time is better than Slow_Write, so replicas with Fast_Write catch up quickly. It is observed that as $fast_{r1}$ increases the age of Fast_Read_0 and Fast_Read_1 increase slightly due to the load on the resources. But the overall age decreases as the age of Fast_Read_1 dominates. It is observed however, that the age is only a fraction of an update and that the difference between the age of reads of the two writes ranges from 0.25 to 0.5 updates.

In conclusion, with Fast_Write every performance metric is better than with Slow_Write except the age. However, the loss in age is considered to be low and the benefits appear to be much higher than the losses.

As before, it is observed that Fast_Read_1 offers better age in exchange of some loss in the response time. For instance, for an even mix of Fast_Read_0 and Fast_Read_1, Fast_Read_1 offers

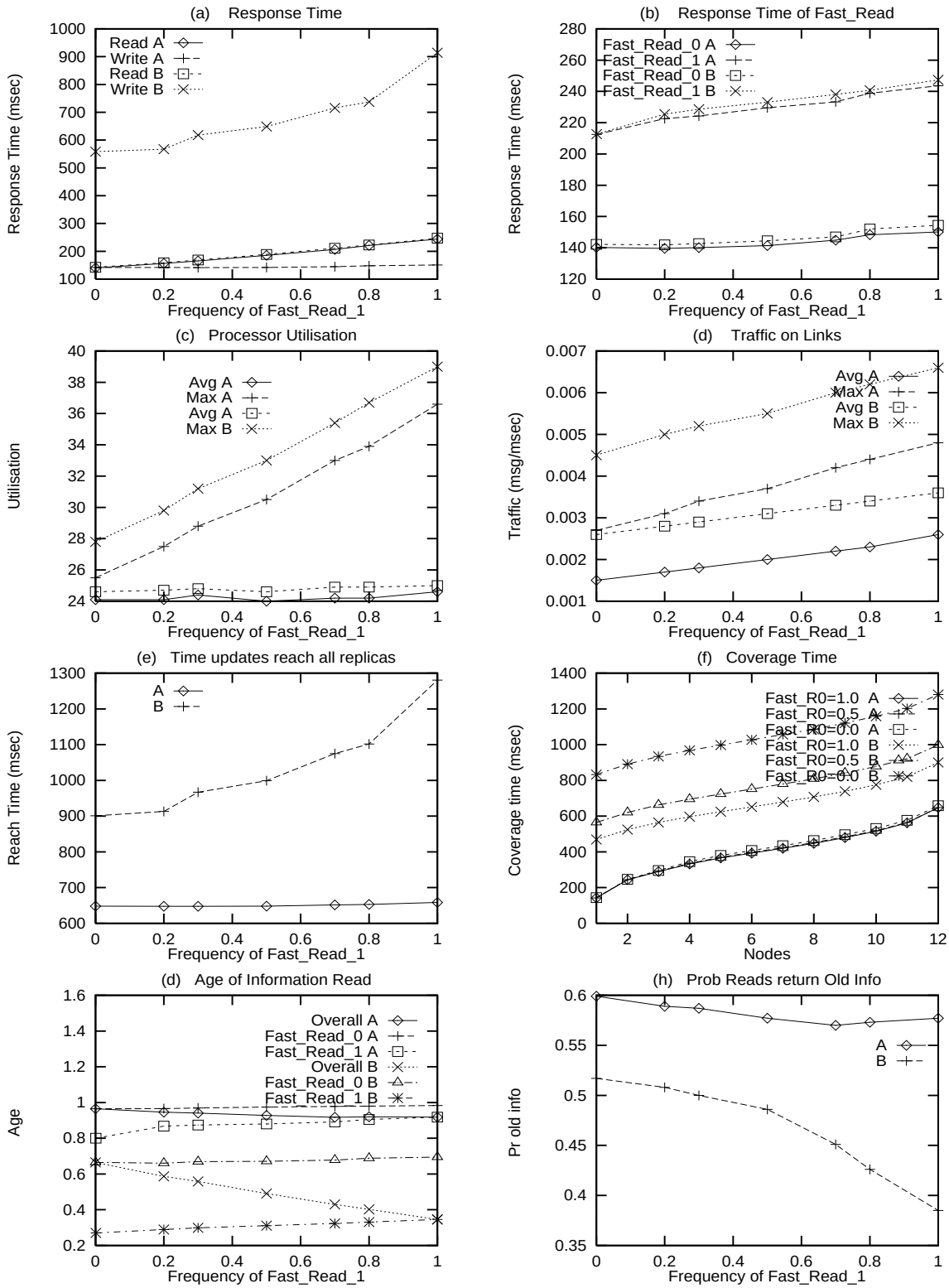


Figure 17: Comparing Slow_Write versus Fast_Write with a mix of Fast_Reads

an improvement in the age of 10% for load A and 54% for load B, with a degradation in the response time of 60%.

### 4.3.2 Varying the arrival rate

This experiment explores the sensitivity of asynchronous operations to the load intensity and compares it to that of synchronous operations. Therefore, it compares a load (A) consisting of Fast_Write and Fast_Read with a load (B) where updates are a mix of Slow_Write and Opt_Write and reads are a mix of Slow_Read and Fast_Read while the arrival rate is varied. Parameters setting is as follows:

| Exp | $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | *inter_arr_rate* |
|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0.5 | 0.5 | 0 | 250-1300 msec |
| B | 0.5 | 0.5 | 0 | 0.25 | 0.25 | 0.5 | 250-1300 msec |

Results are plotted in Figure 18. For load A, it observed that the response time of Fast_Write, Fast_Read_0 and Fast_Read_1 (Figure 18(a)) follows an exponential curve with varying the arrival rate. Fast_Read_0 and Fast_Write have nearly the same response as they involve only processing. While Fast_Read_1 has higher response since it is granted from the top cluster nodes which are more overloaded and also it involves communication. We notice that the increase in the response time is quite slow for low rate. The response of Fast_Read_0 and Fast_Write increase by less than 80 msec for a range of *inter_arr_rate* between 1300-400 msec, and the response of Fast_Read_1 increases by less than 170 msec. For higher rates, the increase is quite sharp. This is due to the high load on the resources, as shown in Figures 18(b) and (c), which are saturated at *inter_arr_rate* $<$ 250 msec. Hence, the bottleneck here comes from resource contention. Similarly, we find that the time at which updates reach replicas (Figure 18(d)) increases slowly ( less than 300 msec for a range of *inter_arr_rate* between 1300-400 msec) then it increases sharply again due to resource contention. Consequently, the age (Figure 18(f)) follows the same curve as the reach time. Again the increase is quite slow until the point 400 msec after which it increases sharply. Further, the age of Fast_Read_0 is higher than that of Fast_Read_1 by 1 to 4 updates for high loads (*inter_arr_rate* $<$ 300 msec) then the difference decreases and drops down to an average of 0.02 updates for a range of *inter_arr_rate* between 1300-500 msec. The increase in the age can also be explained by Figure 18(e) where we can see that the coverage of nodes is quite quick for moderate loads, then it becomes slow for high loads.

However, in comparison with load B, it is observed that with asynchronous operations we can achieve a much higher throughput than with synchronous (2.8 times higher). Further, the performance is much better with asynchronous operations. For instance, for a point with *inter_arr_rate* = 700 msec, the response time of writes is 12.7 times lower, the response time of reads is 7 times lower and the reach time is 5 times lower. Even the age returned by Fast_Read is better: 0.682 versus 2.563 due to the quick coverage and low reach time associated with Fast_Write.

So, in conclusion we can say that with Fast_Write and Fast_Read we can get better performance than with a mix of Slow_Write and Slow_Read, especially with high loads and we can achieve a better throughput.

Figure 19 shows the results of the same experiment for load A but with $freq_r$=0.7. It is thought that such a mix would be more appropriate for these kinds of operations (Fast_Write) which allow for more updates to be done. The figures are plotted and the same comments apply

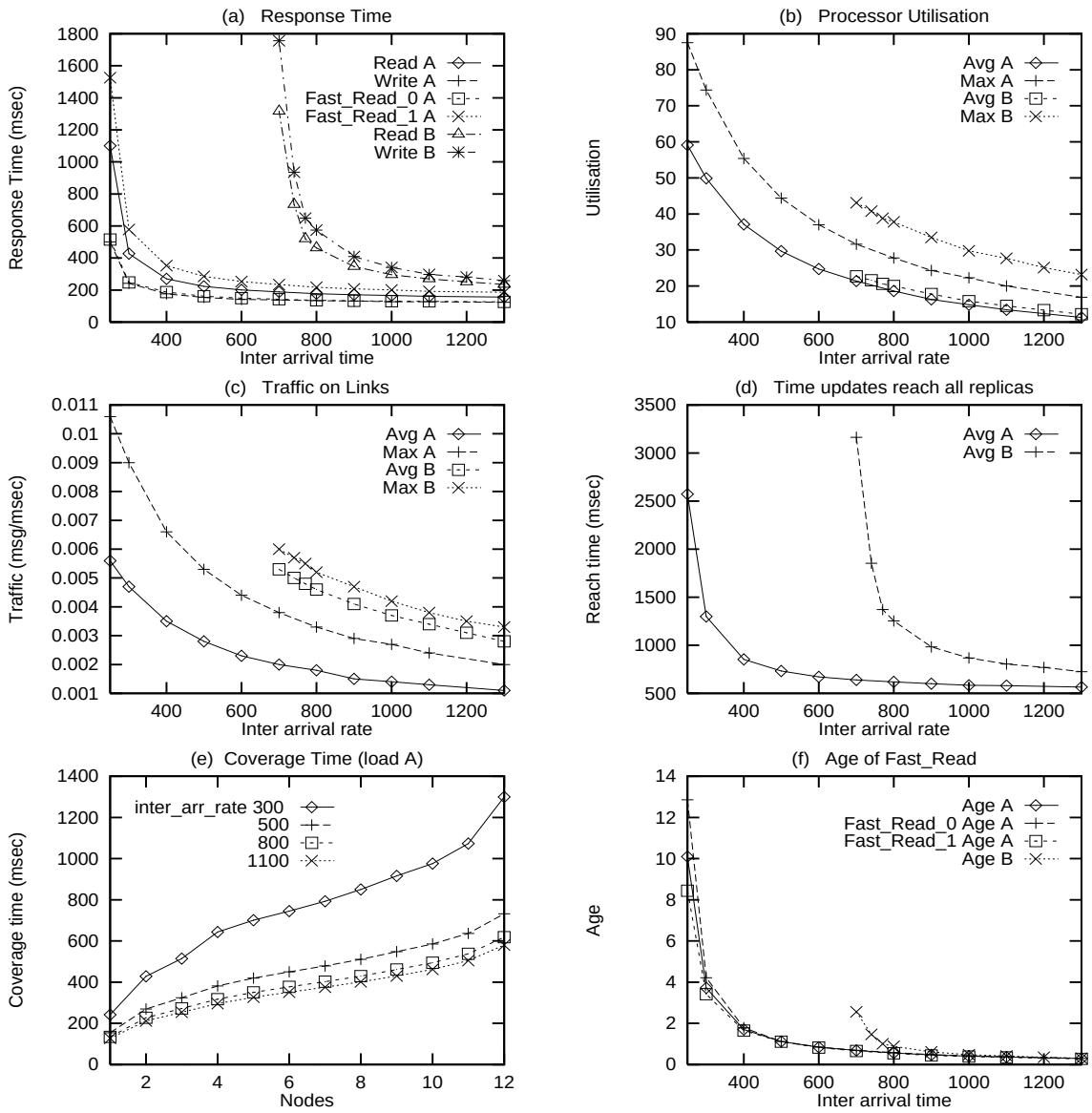


Figure 18: Varying arrival rate, writes are Fast_Write, $freq_r$=.85

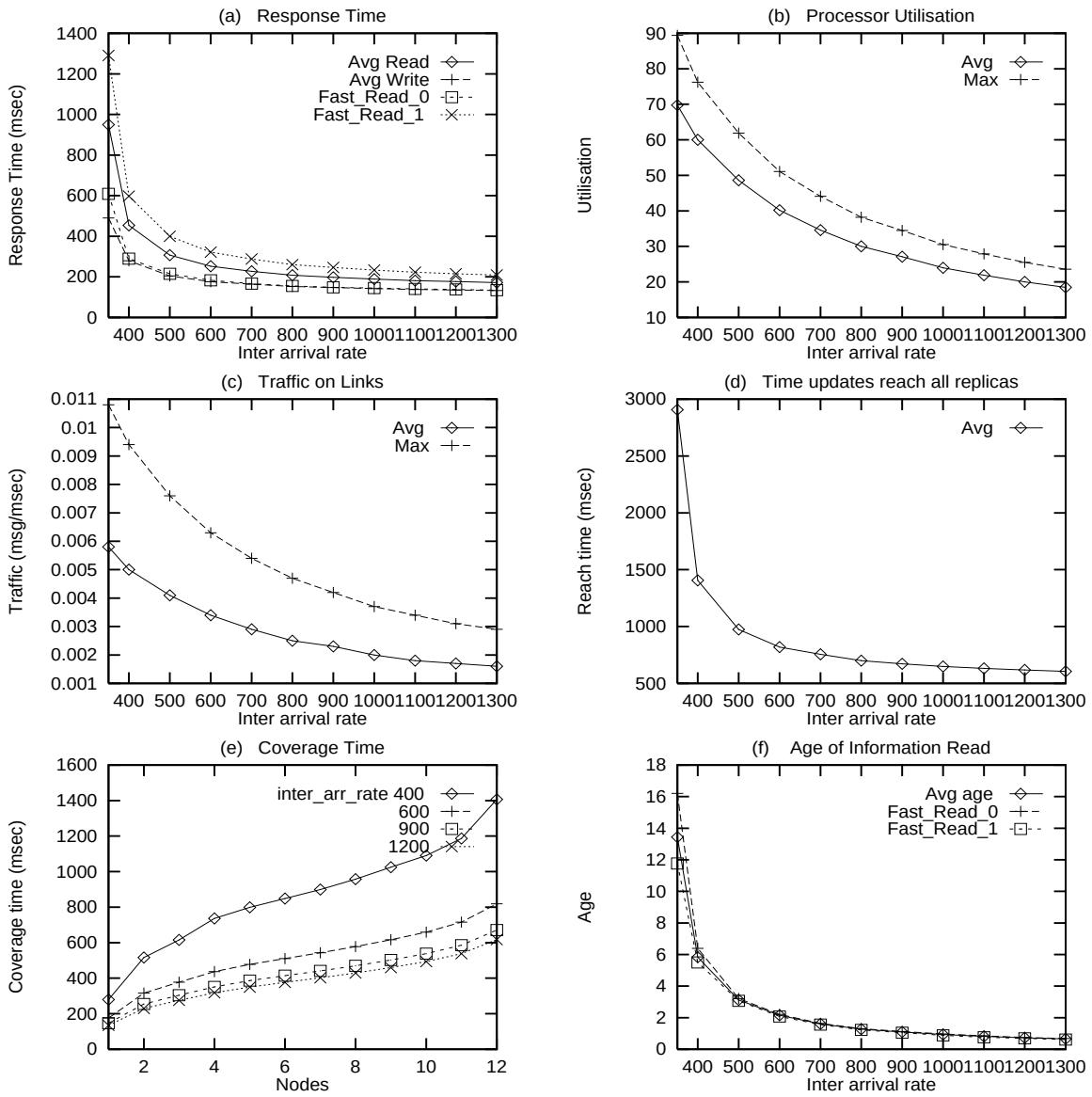


Figure 19: Varying arrival rate, writes are Fast_Write, $freq_r$=.7

as above. Obviously, the resources are more contended and saturation is reached quicker (at 350 msec) since updates place more load on the nodes since every node has to execute all updates.

### 4.3.3 Varying the mix between reads and writes

In this experiment we vary the frequency of reads versus writes to see how the system behaves under different mixes. *inter_arr_rate* is set to 600 msec in order to have a moderately loaded system. Parameters setting is as follows:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | *inter_arr_rate* | $freq_r$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0.5 | 0.5 | 0 | 600 | 0.1-0.9 |

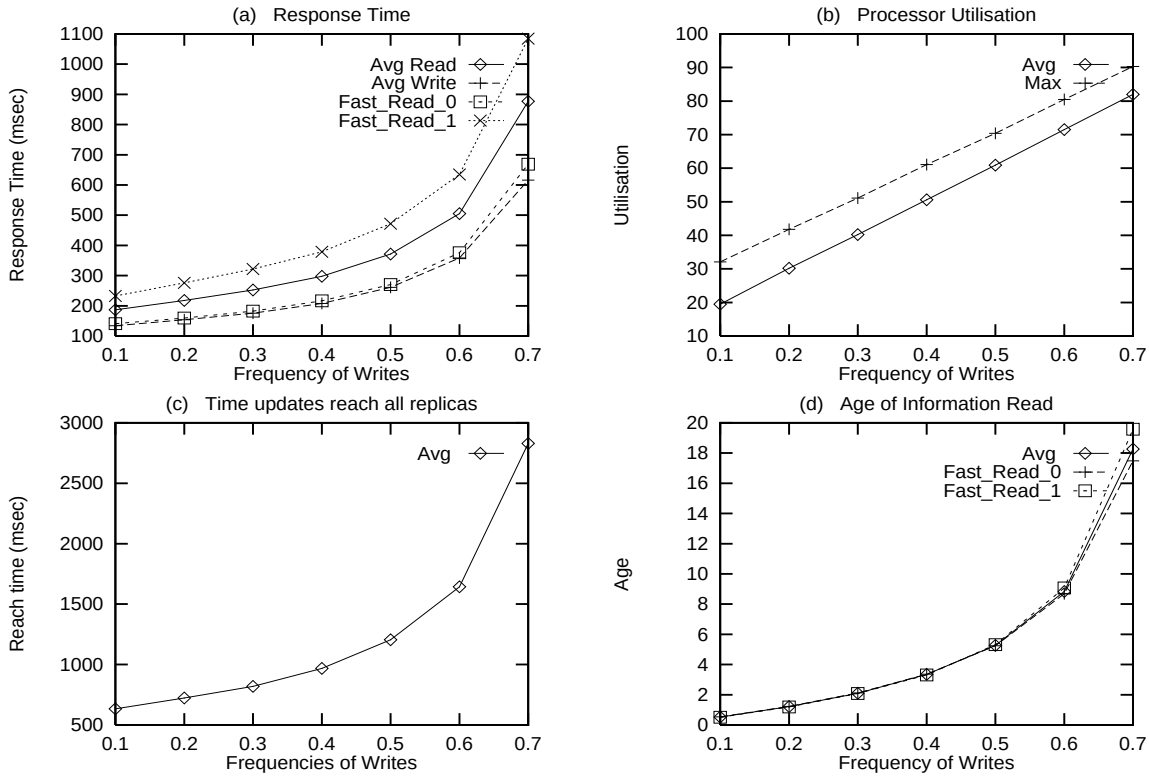


Figure 20: Varying read mix, writes are Fast_Write

As shown in Figure 20(a), as $freq_r$ decreases, the response time increases nearly exponentially. The time by which updates reach other replicas (Figure 20(c)) follows also an exponential curve and so does the age returned by Fast_Read (Figure 20(d)). The reason is that writes place more load on the resources, as shown in Figure 20(b), since every node has to execute the update. Also, they place more load on communication lines while propagating.

If we compare this experiments with its counterpart experiment when writes are synchronous (Figure 7): although it is not a very fair comparison since for that experiment *inter_arr_rate* was set to 1000 while here it is 600 i.e. the load is nearly doubled. However, we see that

with Fast_Write we can achieve a much higher write throughput than with synchronous writes: saturation is reached at $freq_r$<0.3 while with synchronous writes saturation was reached at $freq_r$<0.73. So the maximum write throughput with synchronous writes was 0.0032 while with Fast_Write it is 0.014 updates/msec which is 4.4 times greater. Further, if we compare the two experiments for the same frequency of writes, say 20%, we find that Fast_Write has better performance than synchronous writes even though the load is nearly the double. The average response of writes is 154.20 versus 462.9, response of reads is 217.78 versus 392.00 and the reach time is 722.84 versus 1067.12. However the age is 1.216 versus 0.809; but this is the consequence of having larger number of updates (nearly doubled) which increases the age.

In conclusion, we can say that for application that have high update rates, then Fast_Write is more suitable than synchronous writes, as long as the application can reconcile conflicting updates, since it offers better performance and throughput.

#### 4.3.4 Varying the communication processing overhead

In this experiment, we explore the sensitivity of the performance to the communication processing overhead (*cpu_msg*). Parameters are set as follows:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | *inter_arr_rate* | $freq_r$ | *cpu_msg* |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0.5 | 0.5 | 0 | 600 | 0.7 | 0-20 |

As shown in Figure 21(a), the average response time of reads and writes increases as *cpu_msg* increases. For *cpu_msg*=10 msec, there is an increase of 25.4% for read response, 10.3% for write response, 33.4% in reach time and 49% in the age. The reason behind the degradation is the increase of the processor utilisation (Figure 21(b)) which is due to the increase in the communication overhead as shown. Obviously, this creates more contention on the processors which increases the response time and slows down the speed of propagation. As shown, the response times of Fast_Read_0 and Fast_Write suffer the least as they do not involve communication; so they just suffer from the contention for the processors. Fast_Read_1 is the most affected since it involves communication.

However, it is noticed that Fast_Write, Fast_Read_0 and Fast_Read_1 are less sensitive to *cpu_msg* than Slow_Write, Opt_Write and Slow_Read. Comparing with the experiment reported in Figure 8, although the load is much less than this experiment (*inter_arr_rate*=1000 versus 600 and $freq_r$=0.85 versus 0.7); for a *cpu_msg*=15 msec; synchronous operations place an average of communication overhead of 14.71% and a maximum of 34.5% while asynchronous have an average of 16.0% and a maximum of 30.6%.

Figure 22 reports results of an experiment set to compare the behaviour of asynchronous and synchronous operations under the same load while varying the communication processing overhead (*cpu_msg*). The parameters setting is as follows:

| Exp | $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | *cpu_msg* |
|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0.5 | 0.5 | 0 | 0-15 msec |
| B | 0.5 | 0.5 | 0 | 0.25 | 0.25 | 0.5 | 0-15 msec |

As shown in Figure 22(a), the average read and write response times of load A are far better than those of B. The difference is small for small values of *cpu_msg* then it increases as *cpu_msg* increases. The reason is due to that the communication overhead that load B places on the processors is much more than in A as shown in Figure 22(b). Consequently, the time at which

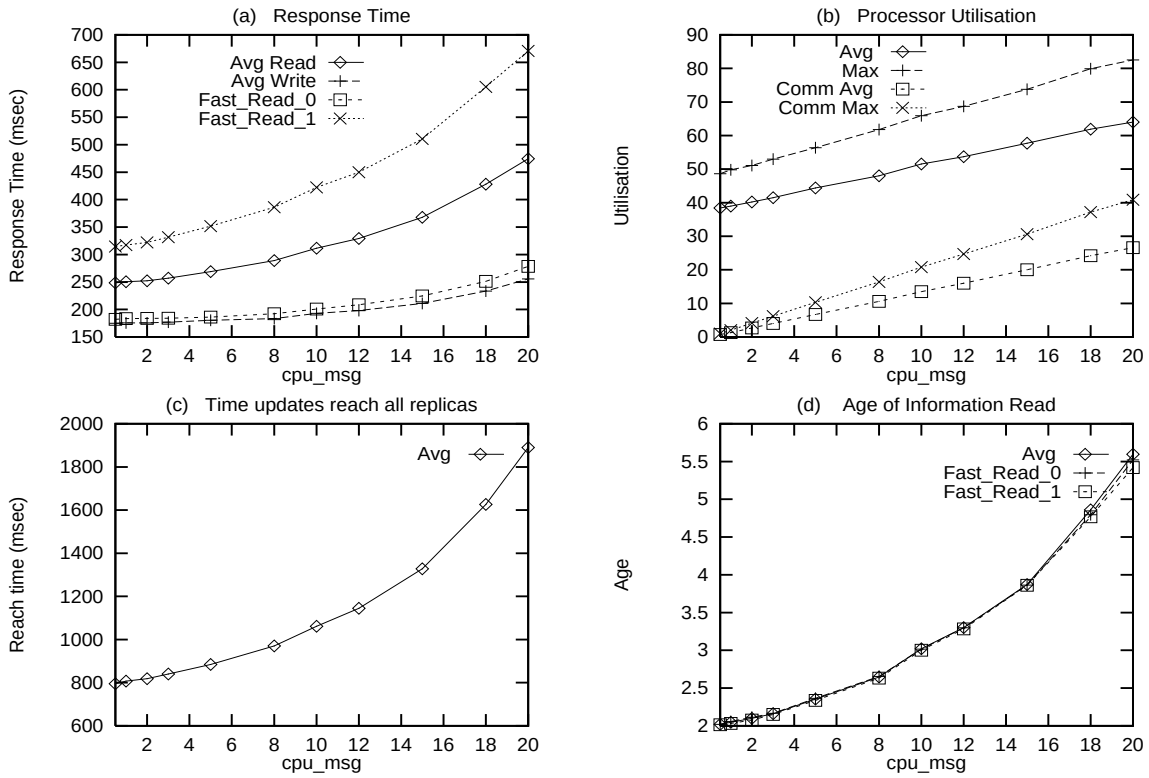


Figure 21: Varying *cpu_msg*, writes are Fast_Write

updates reach replicas in A is much better than B (Figure 22(c)). Although it increases with the increase of *cpu_msg*, the increase is very small in comparison to that of B. It is interesting to see that although load B contains mostly synchronous updates, the age returned by Fast_Read in A is better than B, especially for high values of *cpu_msg*, as depicted in Figure 22(d).

From here one can say that fast (asynchronous) operations are less sensitive to communication overhead than are slow (synchronous) operations and they perform better, especially under systems with high communication overhead.

### 4.3.5 Varying the network delay

In this experiment, we explore the sensitivity of performance to the delay on the network (*wan_del*). *lan_del* is fixed to 0.02 msec and *wan_del* is varied from 0.2 to 100 msec. Parameters setting is as follows:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | *inter_arr_rate* | $freq_r$ | *wan_del* |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0.5 | 0.5 | 0 | 600 | 0.7 | 0-100 |

As shown in Figure 23(a), the response time of Fast_Read_0 and Fast_Write are not affected since they do not involve communication. The response of Fast_Read_1 increases slowly with low values of *wan_del*. The increase is around 16% to 25% for *wan_del*= 20 to 30 msec. Then,

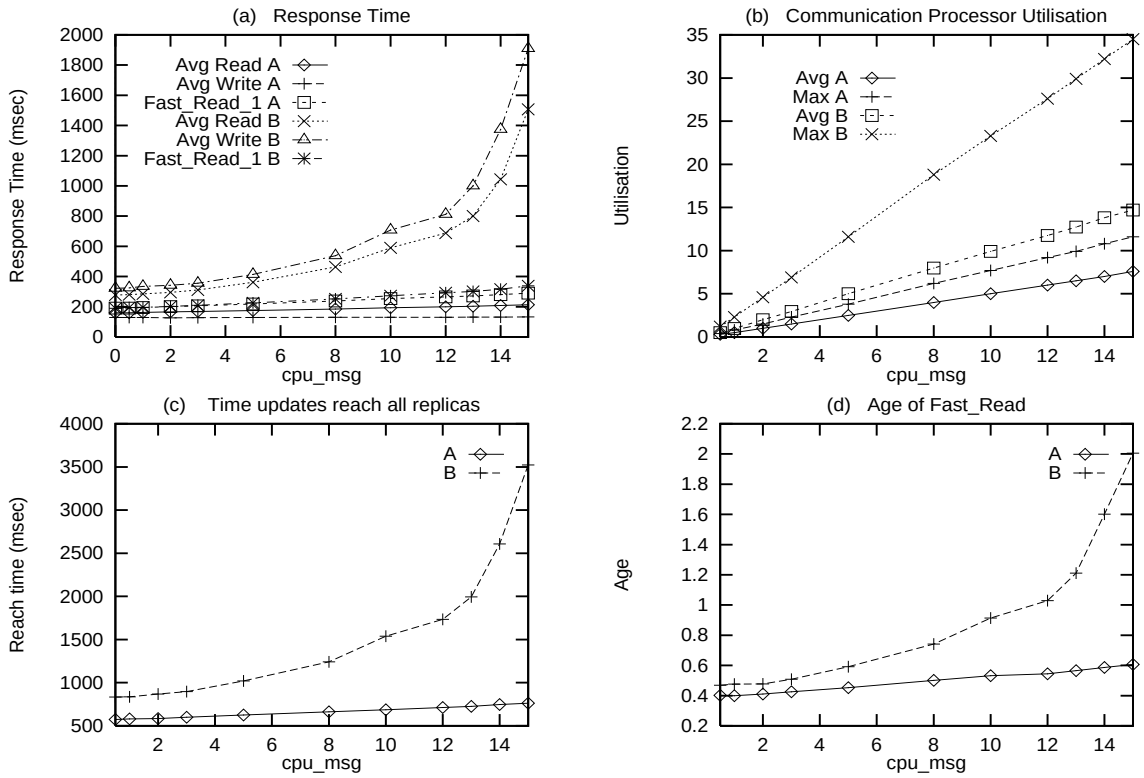


Figure 22: Varying *cpu_msg*, comparing asynchronous and synchronous operations

it increases more sharply (up to 46% and 138.7% for *wan_del* = 50 and 100 msec), when the transceivers become more contended [1]. Similarly, the reach time increases, as shown in Figure 23(b), by 5.7%, 20.6% and 70.6% for *wan_del*=20, 50 and 100 respectively. The age increases as well with *wan_del* since updates take longer to propagate to all replicas. The age returned by Fast_Read_0 suffers more than that of Fast_Read_1 as *wan_del* increases. However, the increase of the overall age is around 1.5 updates for *wan_del*=100 msec.

In Figure 24, we compare the performance of synchronous versus asynchronous operations while varying *wan_del* to see how sensitive they are to it. Parameter setting is as follows:

| Exp | $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | *inter_arr_rate* | $freq_r$ | *wan_del* |
|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0.5 | 0.5 | 0 | 1000 | 0.85 | 0-100 msec |
| B | 0.5 | 0.5 | 0 | 0.25 | 0.25 | 0.5 | 1000 | 0.85 | 0-100 msec |

As shown in Figure 24(a), the average read and write responses of load A are far better than B. Response of Fast_Read_1 is higher with load B (around 10-20 msec) due to higher load on processors and transceivers. As shown in Figure 24(b), the reach time of both A and B increases but the increase of B is much sharper than that of A. It is interesting to see that the age returned by Fast_Read with B is much higher than that returned by A, especially for high values of *wan_del*. This is due to the huge difference in the time at which updates reach all

[1] the comparison is w.r.t. the ideal case when *wan_del*=0

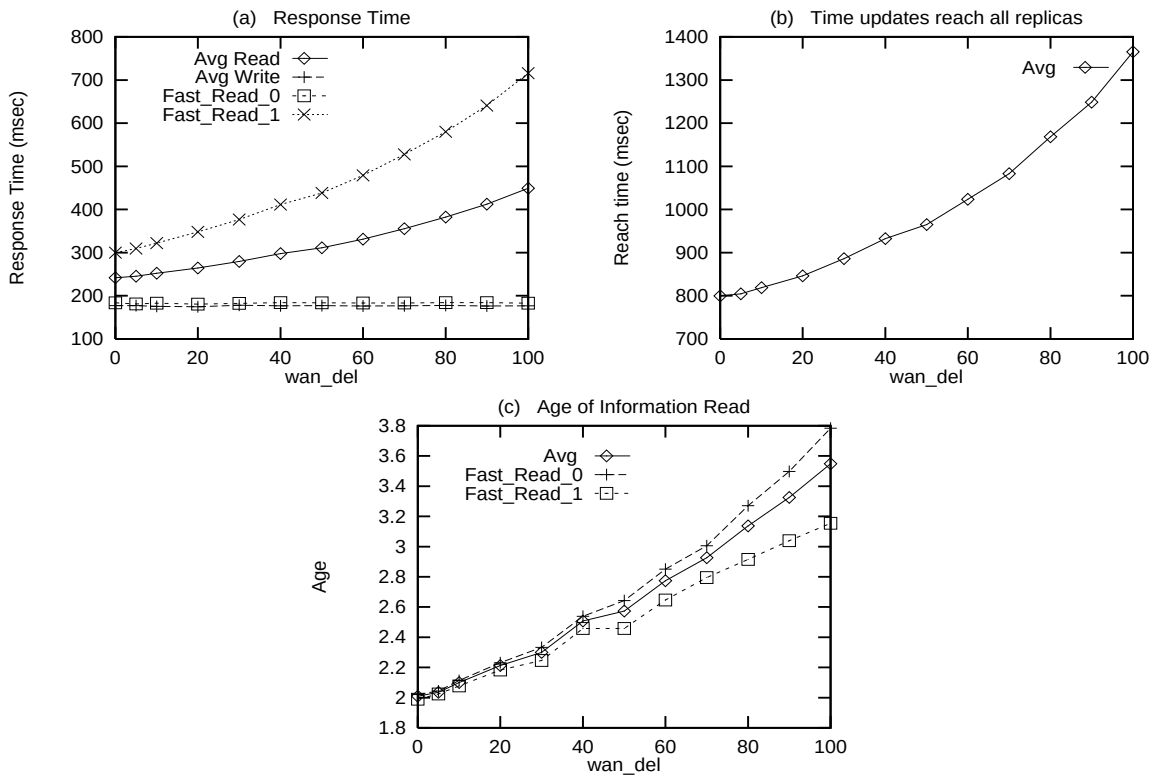


Figure 23: Varying the network delay, writes are Fast_Write

replicas. So, from here we can deduce that load A is far less sensitive to *wan_del* than load B. Therefore, asynchronous operations can cope with higher values of *wan_del* and still offers good performance.

## 4.4 Varying network topologies when updates are Fast_Write

This section explores how different hierarchical network topologies can affect the performance of asynchronous operations. The same set of networks shown in Figure 11 is considered. For this set of experiments, the read mix between Fast_Read_0 and Fast_Read_1 is varied when updates are Fast_Write and the results compared to those of Net 1. Since asynchronous operations can cope with higher load than synchronous operation, the experiments are set to generate a higher load than the default. The following setting is considered:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | *inter_arr_rate* | $freq_r$ | *cpu_msg* | *wan_del* |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0-1 | 0-1 | 0 | 600 msec | 0.7 | 5 msec | 30 msec |

### 4.4.1 Network Net 2

The average response times of reads and writes of both Net 1 and Net 2 are plotted in Figure 25(a). The average response time of writes increases slightly as more Fast_Read_1 requests are introduced. This is because Fast_Read_1 places more load on specific nodes (1,2 and 3 in

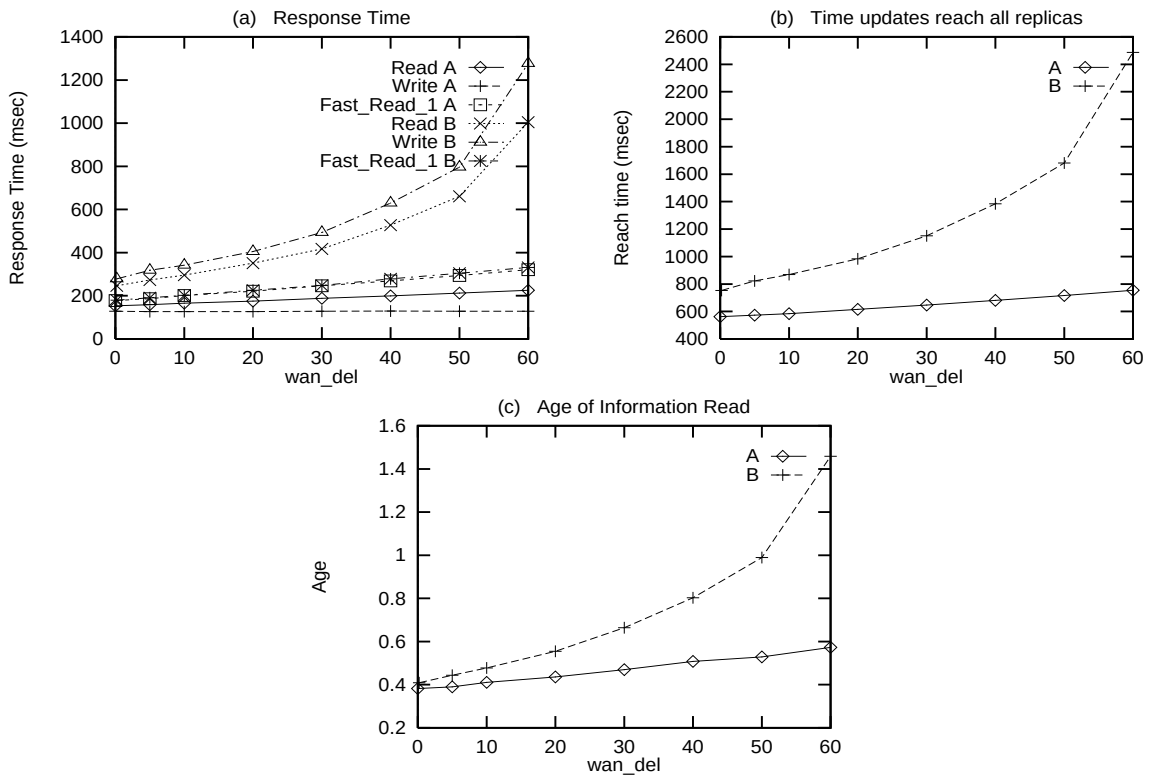


Figure 24: Varying the network delay, comparing synchronous and asynchronous operations

Net 1 and 3,5 and 6 on Net 2) as $fast_{r1}$ increases and hence creates more contention (Figure 25(c)). The average read response increases more sharply as shown due to the high response of Fast_Read_1. The response of Fast_Read_0 (Figure 25(b)) follows the same increase as Fast_Write while that of Fast_Read_1 increases more sharply.

Comparing Net 1 and Net 2, we observe that the response time of all operations (Fast_Write, Fast_Read_0 and Fast_Read_1) are the same for low frequency of Fast_Read_1 but are slightly lower in Net 2 than Net 1 for high frequency of Fast_Read_1 (around 15-25 msec lower). The reason for this is that Fast_Read_1 on Net 2 places less contention on the nodes in the top cluster since the requests are granted from nodes 3,5 and 6 instead of 1,2 and 3. In Net 1 nodes 1,2 and 3 have to satisfy their own Fast_Read_0 and Fast_Read_1 requests as well as Fast_Read_1 of their children. While in Net 2, only node 1 has to do so while node 2 and 3 have to satisfy their own Fast_Read_0 and Fast_Read_1 requests only. This distributes the load more evenly among most of the nodes, consequently the response time decreases. The time by which updates reach all replicas (Figure 25(d)) increases as $fast_{r1}$ increases since Fast_Read_1 places more load on the resources. However, it is nearly the same for both networks since each update has to traverse the same number of hops. As shown in Figure 25(e), the age starts to decrease as $fast_{r1}$ increases since Fast_Read_1 have less age than Fast_Read_0. Then, at $fast_{r1}$>0.5, the age starts to increase. The reason is: as more Fast_Read_1 are present, more contention is placed on the resources which increases the time by which updates reach other replicas and deteriorates the age. We notice that the age of

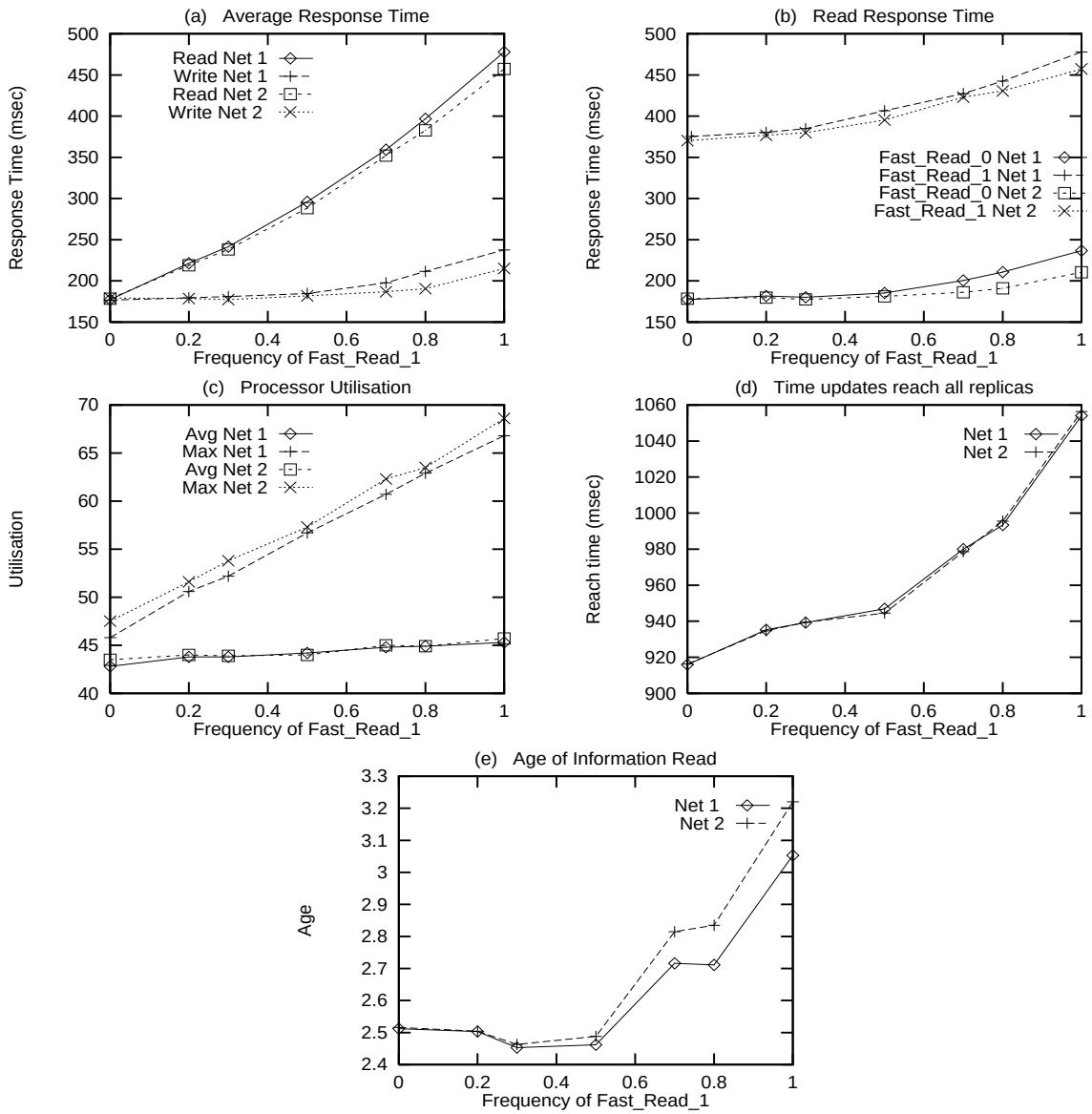


Figure 25: Varying read mix, Network 2

Net 1 and Net 2 are nearly the same for low frequency of Fast_Read_1, then the age of Net 2 increases and is higher than Net 1 (around 0.1-0.15 updates higher). The reason is that in Net 1, nodes 1,2 and 3 are the most up-to-date nodes and all Fast_Read_1 requests are satisfied from them. While in Net 2, nodes 4,5 and 6 are the most up-to-date but only half of the Fast_Read_1 requests are satisfied from them.

In conclusion, for Fast_Write and Fast_Read operations, there is not much difference in performance between Net 1 and Net 2. Net 2 offers a slightly better response time while Net 1 offers a slightly better age; but the difference is not big.

### 4.4.2 Network Net 3

Results of Net 3 are quite similar to Net 2. The response time of Fast_Write, Fast_Read_0 and Fast_Read_1, shown in Figures 26(a) and (b), are nearly the same for both networks for low $fast_{r1}$, but they are lower in Net 3 for high $fast_{r1}$ (about 15-25 msec lower). The reason is the same as for Net 2: on Net 1, Fast_Read_1 places more load on nodes of top cluster while on Net 3 it spreads the load of Fast_Read_1 on nodes 1,2, 3, 5 and 9. So the load is more balanced which improve the response time.

The time by which updates reach replicas (Figure 26(d)) is higher in Net 3 by an amount of 35-40 msec. This is because Net 3 consists of 4 levels; so it takes updates originated at a root or a leaf node 7 hops (4 wan links) to reach other replicas as opposed to 5 hops (3 wan links) in Net 1. Consequently, the age returned by Fast_Read is higher in Net 3 than in Net 1, as shown in Figure 26(e). The difference is small for low $fast_{r1}$ (around 0.05 updates) then it increases to 0.25-0.35 updates for high $fast_{r1}$. There are two reasons for the increase. First, since the reach time is higher, then replicas in Net 3 are less up-to-date than in Net 1. Second, from the construction of the Net 3, the most up to date nodes are the nodes in the centre of the tree i.e. 4,5,6,7,8 and 9. However, Fast_Read_1 of half of the nodes (1,2,3,4,5 and 6) are satisfied from nodes in the top cluster which are not the most up to date. While in Net 1, Fast_Read_1 are all satisfied from nodes 1,2 and 3 which are the most up-to-date.

It is believed that the benefit gained in the response time with Net 3 is less than the losses in the speed of propagation and the age in comparison with Net 1.

### 4.4.3 Network Net 4

As shown in Figure 27(a), the average response time of writes increases as $fast_{r1}$ increases for both networks. Both Net 1 and Net 4 have similar response, with Net 4 slightly higher than in Net 1 for high $fast_{r1}$. The reason is that Net 4 relieves nodes on the top cluster from executing all Fast_Read_1 requests; but instead of distributing it evenly it transfers the load of nodes 2 and 3 to node 5 which creates more contention on node 5. As shown in Figure 27(c), the average utilisation is the same for both networks; but the maximum utilisation, which represents the load on node 1,2 or 3 for Net 1 and the load on node 5 for Net 4, is much higher on Net 4 since most of Fast_Read_1 are granted from node 5. The average response of read (Figure 27(a)) is nearly the same for both networks at low $fast_{r1}$, then it increases sharply on Net 4 with an increase of 180 msec (38%) for $fast_{r1}$=1. The response time of Fast_Read_0 and Fast_Read_1 are plotted in Figure 27(b). Fast_Read_0's response is the same as for Fast_Write, since it involves only processing. For Fast_Read_1, it is constantly higher in Net 4 than in Net 1 and the difference

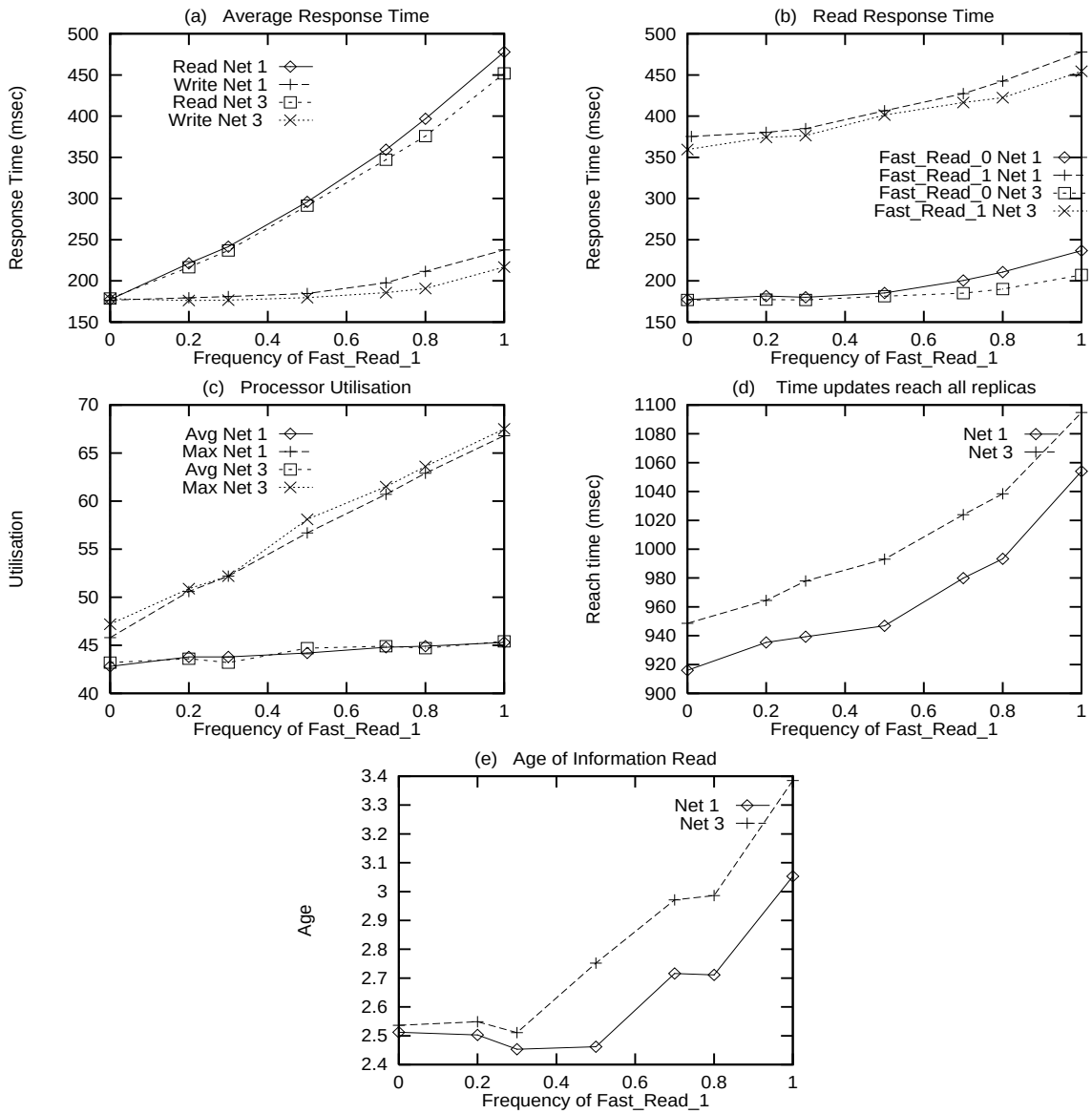


Figure 26: Varying read mix, Network 3

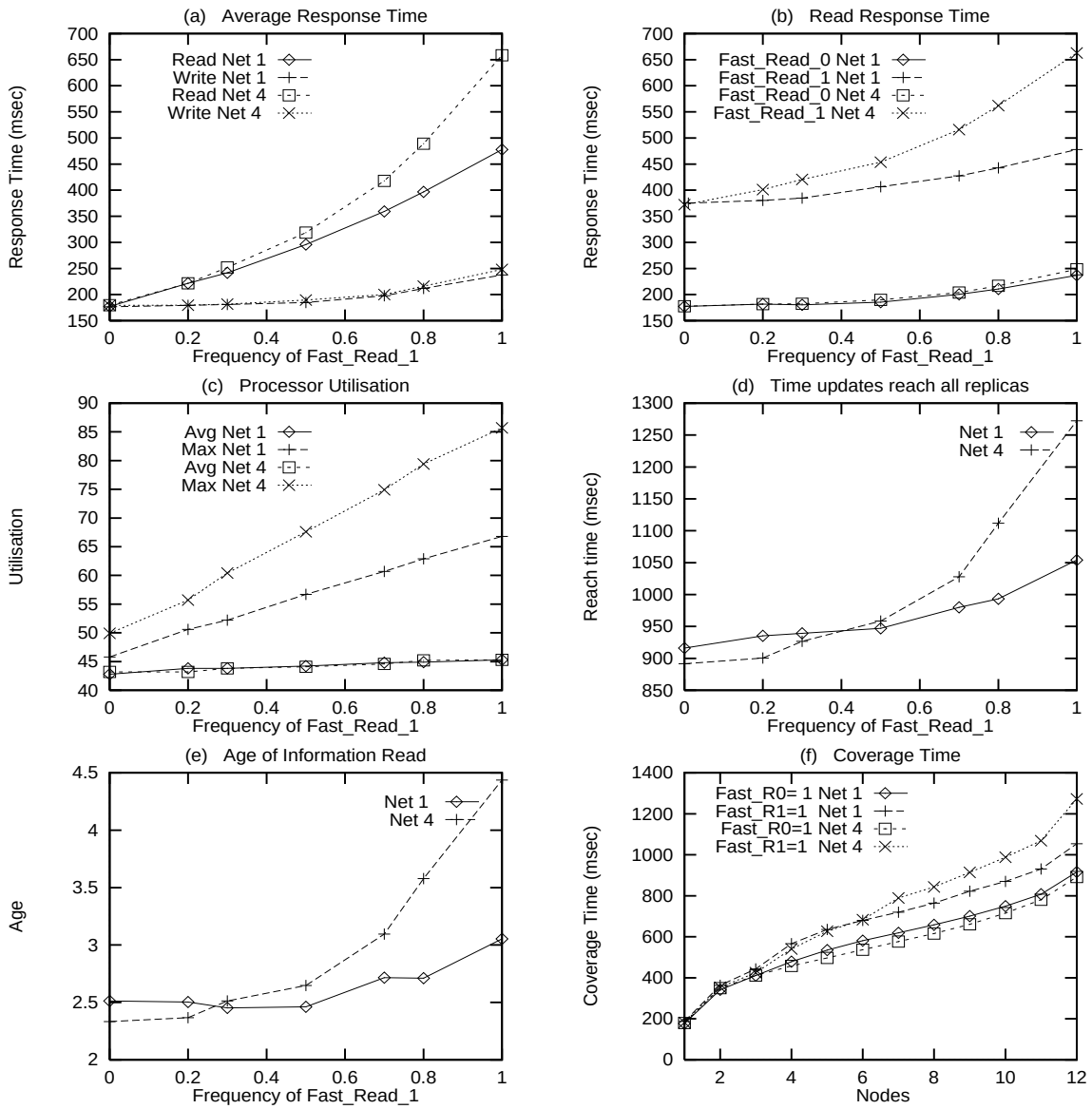


Figure 27: Varying read mix, Network 4

increases as $fast_{r1}$ increases. The main reason for the increase is the extra load Fast_Read_1 places on node 5 in Net 4.

The same reasons cause the reach time to increase sharply in Net 4, as shown in Figure 27(d). It is lower than Net 1 for low $fast_{r1}$, then it increases sharply to a difference of 218 msec (20.7% higher) when $fast_{r1}$=1. Consequently, the age follows the same curve (Figure 27(e)), it is slightly lower than Net 1 for low $fast_{r1}$, then it increases up to a difference of 1.38 updates (45.3% higher) for $fast_{r1}$=1.

It was interesting to observe the coverage time -speed of propagation- for the two networks. It is plotted in Figure 27(f) for 2 points when $fast_{r0}$=1 and $fast_{r1}$=1. For $fast_{r0}$=1, the coverage time in Net 4 is better than in Net 1. This is due to that nodes in the bottom cluster are directly connected and updates generated at any of them is spread faster than having to go through parents. This explains the benefit in age for low $fast_{r1}$. For $fast_{r1}$=1, we see the coverage time is slightly better (around 10 msec lower) for the first few nodes, then it increases sharply. This is due to the contention on node 5 and the communication lines.

In conclusion, performance of Net 4 is slightly better than Net 1 for low frequency of Fast_Read_1 (up to 20%); but for higher values performance degrades in all metrics and is much worse than Net 1.

### 4.4.4 Network Net 5

On this network, the performance is worse than on Net 1 for all read mixes and for all metrics. The reasons are similar to those of Net 4. First, there is an extra load placed on node 1:

(a) it is acting as the parent of most nodes, hence it is granting their Fast_Read_1 requests. This places more processing overhead on the node.

(b) node 1 is the communication link between nodes 2,3 and the rest of the nodes, hence an extra communication processing overhead is placed on it.

(c) since it is the link between nodes 1,2, and the rest of the nodes, there is an extra propagation overhead placed on it.

As seen from Figure 28(c), the maximum load on processors (which is the load on node 3) is constantly higher in Net 5 than that of Net 1. For low $fast_{r1}$, it is due to factors (b) and (c). For high $fast_{r1}$ it is due to the 3 factors. It should be noticed that processors reached saturation for $fast_{r1}$=1 due to the high overhead.

This resource contention causes increase of the response time as shown in Figure 28(a), (b). We notice that every single metric is worse in Net 5 and the degradation is very high for large $fast_{r1}$ when the contention is very high.

Also, we notice that the reach time (Figure 28(d)) is constantly higher than in Net 1. Although updates need to travel a maximum of two hops to reach all replicas but because the contention on the resources (CPU, disks, links) is higher, then the reach time increases. Consequently, the age follows the same curve.

From here we can conclude that Net 5 is by no means better than Net 1 in any aspect.

### 4.4.5 Network Net 6

This experiment shows the performance of asynchronous operations over the fully connected network Net 6 (Figure 15) when the arrival rate is varied and results are compared to Net 1. Since Net 6 involves 15 WAN links while Net 1 involves only six, assuming that lines of Net 6

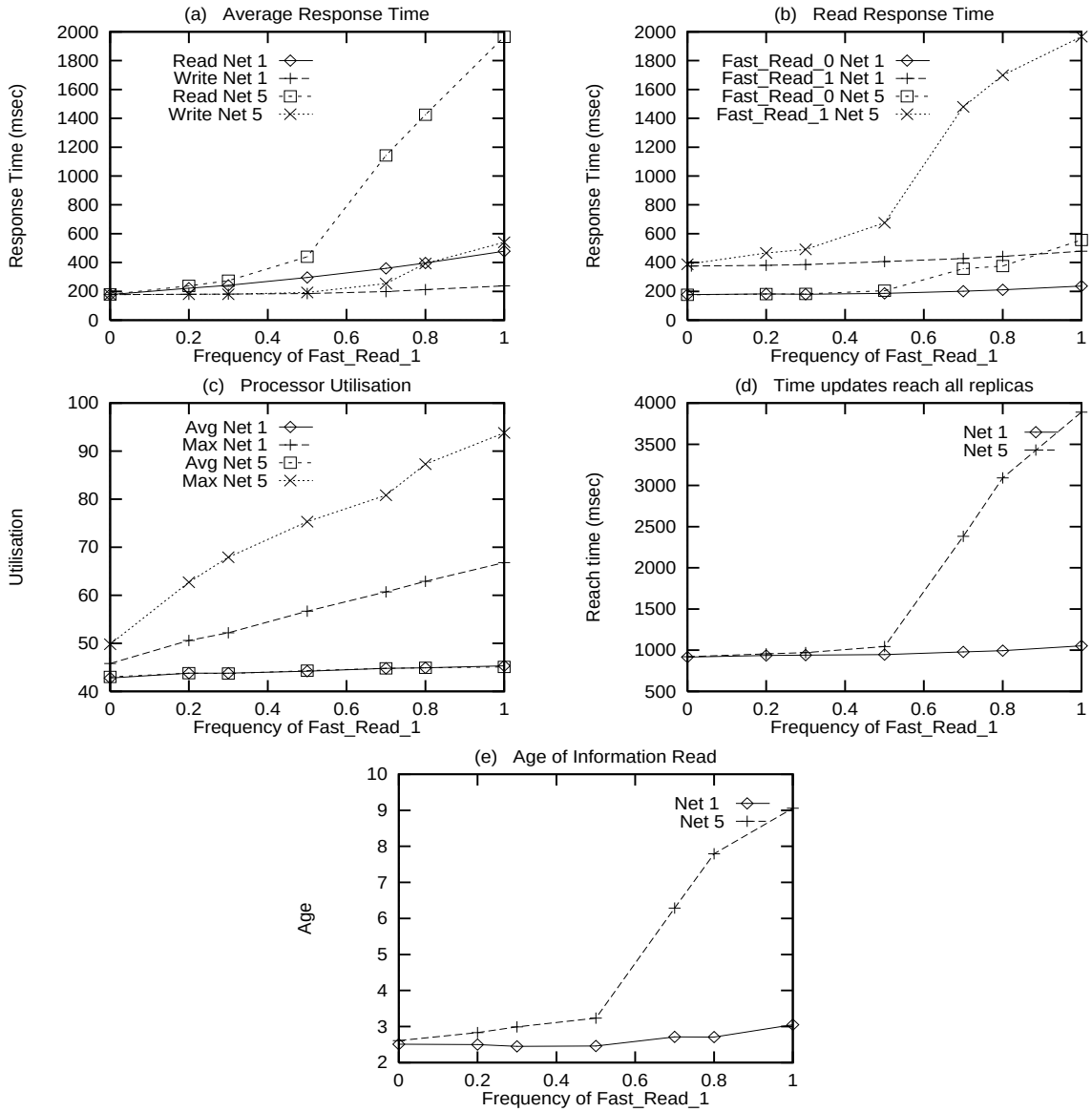


Figure 28: Varying read mix, Network 5

have the same bandwidth as Net 1 seems unfair, if economical constraints are taken into account in the comparison. Therefore, two versions of Net 6 are considered. The first one (Net 6_A), assumes the bandwidth per line is equal to those of Net 1. The second one (Net 6_B), simulates budget constraints by assuming that the total bandwidth is constant and the bandwidth is reassigned per line accordingly. Therefore, *wan_del* is set to 75 msec for Net 6_B versus 30 msec for Net 1 and Net 6_A. The parameters are set as follows:

| Exp | $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | $freq_r$ | *cpu_msg* | *wan_del* | *inter_arr_rate* |
|---|---|---|---|---|---|---|---|---|---|---|
| Net 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0.5 | 5 msec | 30 msec | 500-1200 msec |
| Net 6_A | 0 | 0 | 1 | 1 | 0 | 0 | 0.5 | 5 msec | 30 msec | 500-1200 msec |
| Net 6_B | 0 | 0 | 1 | 1 | 0 | 0 | 0.5 | 5 msec | 75 msec | 500-1200 msec |

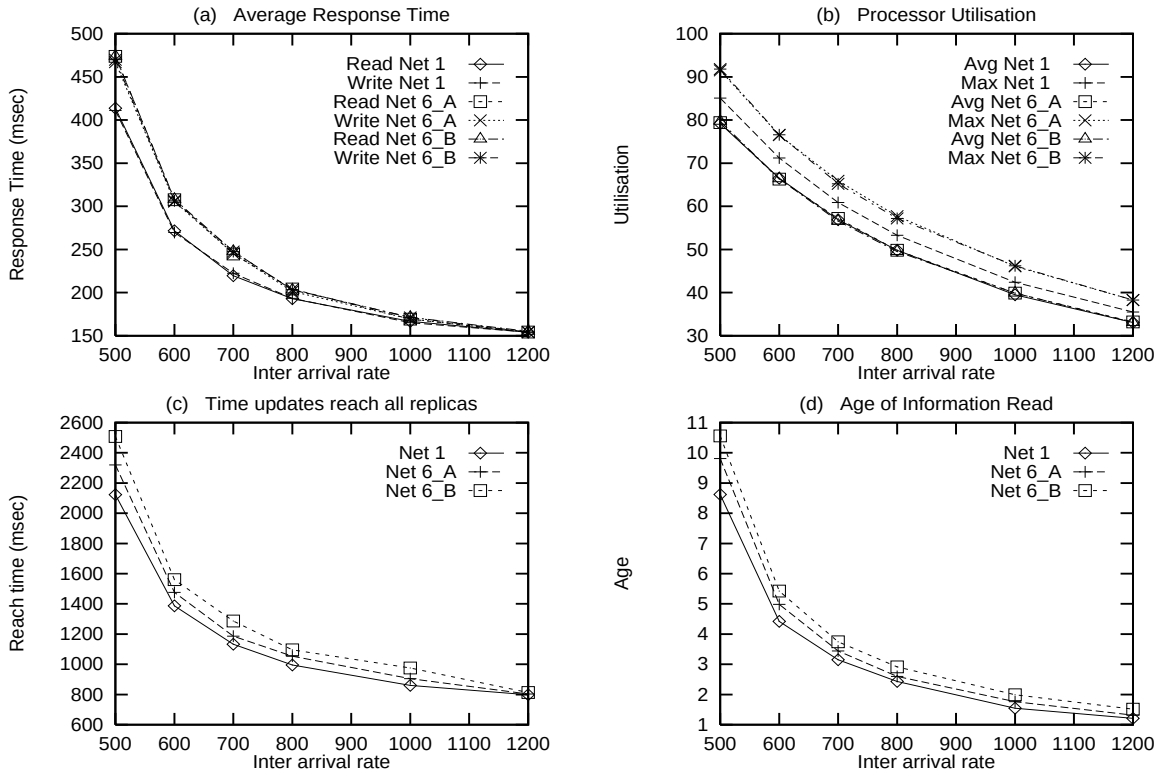


Figure 29: Varying arrival rate, Network 6

As shown in Figure 29(a), the response times for both Net 6_A and Net 6_B are higher than Net 1 with a difference about 30-40 msec for high loads. This is because more processing communication overhead is placed on the nodes in Net 6 due to the extra communication lines. The processing load on the resources of those networks was observed to be constantly higher than Net 1.

For the same reasons, the reach time is higher in Net 6_A than in Net 1. It is the highest in Net 6_B, since the link delays are higher which creates more contention at the communication

lines. Consequently, the age is the best in Net 1, followed by Net 6_A and is the worst in Net 6_B as shown in Figure 29 (b).

Therefore, it can be concluded that the performance of the fully connected network with asynchronous operations is constantly worse than Net 1 especially under high loads.

## 4.5 Networks of 39 replicas, updates are Slow_Write and Opt_Write

These experiments explore the performance over a network consisting of 39 nodes to see how a larger degree of replication can affect performance. Three different physical (and hence logical) structures are considered, as shown in Figure 30. The experiments considers a load consisting of synchronous updates and varying a mix of Slow_Read and Fast_Read. The following setting is used:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | $inter_arr_rate$ |
|---|---|---|---|---|---|---|
| 0.5 | 0.5 | 0 | 0-1* | 0-1* | 0-1 | 3300 msec |

* The ratio between $fast_{r0}$ to $fast_{r1}$ is kept 1:1

### 4.5.1 Network Net 7

From Figure 31(a), it is observed that the average response time of reads and writes on Net 7 decreases as the frequency of Fast_Read increases. From Figure 31(b) we see that the response of Opt_Write is more or less the same. The response of Slow_Write and Slow_Read decrease substantially, as shown in Figures 31(b) and (c), mainly because the time it takes Slow_Write and Slow_Read to commit decreases as shown. The response of Fast_Read_1 (Figure 31(d)) also decreases with decreasing the frequency of Slow_Read. Similarly, the reach time (Figure 31(f)) decreases and consequently the age (Figure 31(g)).

The reasons for the improvement in performance are:

1. The load on resources of the nodes in the top level reduces as Fast_Reads are introduced as shown in Figure 31(e). These nodes perform less communication and 2PC overhead when frequency of Slow_Read decreases and they are exempted from performing all the Slow_Read requests. As $fast_r$ increases, the processing of reads is distributed among the nodes and this balances the load which affects every single metric.

2. The traffic on the communication links decreases, especially in the presence of Fast_Read_0 as they require no communication. This affects the response of Slow_Write, Slow_Read and Fast_Read_1 as well as the reach time and consequently the age.

3. The data contention (lock) decreases as the frequency of Slow_Read decreases, which improves the response time of Slow_Read and Slow_Write, the commit time, and consequently the reach time.

In general, it is observed that with 39 nodes, the limitations placed by synchronous operations are higher than with 12 nodes and the maximum achieved throughput dropped to less than one-third. Further, the age returned by Fast_Read is generally higher with larger number of replicas since updates take longer to reach all nodes.

The same conclusions are reached about Fast_Read_0 and Fast_Read_1: Fast_Read_1 offers better age at the expense of a higher response time. The improvement in the age and the degaradtion in the response time depend on which level the read is performed at.

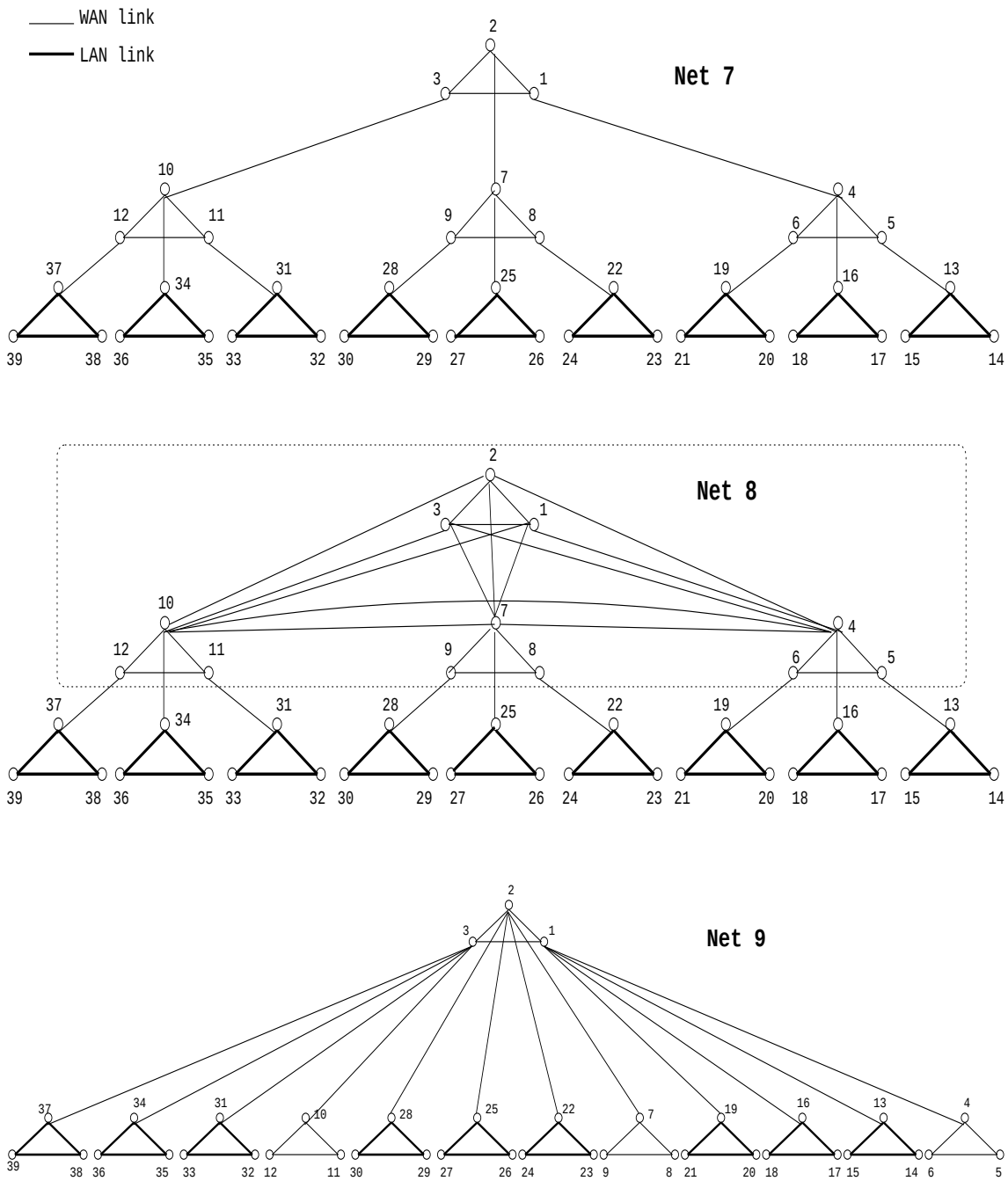


Figure 30: Different network topologies for 39 replicas

### 4.5.2 Comparing Net 7 and Net 8

Comparing Net 8 and Net 7, the results showed that (Figure 31):

- Steady state was not able to be reached on Net 8 for $slow_r$ higher than 0.5. The reasons for this are: since a node needs to assemble a quorum from at least 7 nodes before it commits (versus 2 in case of Net 7), then the commit time increases, the time requests hold locks increase and consequently data contention increases. Further, since nodes are more autonomous: (12 nodes can be a CO as opposed to 3 in Net 7), the rate at which deadlock occurs is much higher. Finally, resources are more loaded since 12 nodes are involved with 2PC and communication overhead as opposed to 3 in Net 7. These factors increase the time by which transactions commit and eventually no steady state could be reached.

- For loads where steady state could have been reached, most of the performance metrics showed a degradation.

  The average response time of writes (Figure 31(a)) is far higher than in Net 7, nearly doubled. Partly, because 30% of Opt_Write (those of nodes in level 1 and 2) are converted to Slow_Write. Further, Slow_Write experiences a higher response time as shown in Figure 31(b) which is mainly due to the commit time. The response of Opt_Write is more or less the same.

  The average response time of reads (Figure 31(a)) in Net 8 is higher than Net 7, especially for low $fast_r$, then it decreases to approach Net 7 for high $fast_r$. As shown in Figure 31(c), Slow_Read experiences a much higher response time in Net 8. It should be noticed that although Net 8 consists of two logical levels which cuts down the communication time to send and receive results to and from CO for Slow_Write and Slow_Read, this does not compensate for the huge increase in the commit time. The response of Fast_Read_0 is higher than Net 7 (Figure 31(d)) especially for high frequency of Slow_Read because the load on the nodes is higher in Net 8 than Net 7 as shown in Figure 31(e). This is due to that nodes 1 to 12 in Net 8 are involved with communication and 2PC overhead while in Net 7 only nodes 1 to 3 are involved with such overheads. Further, since all Fast_Read_1 are granted in Net 8 from nodes 4 to 12, the load on these nodes (which is the maximum) is higher than Net 7 since it involves 2PC and communication overhead, processing Slow_Read, processing its own Fast_Read_0 and Fast_Read_1 as well as processing Fast_Read_1 of nodes in level 3. Consequently, the response of Fast_Read_1 (Figure 31(d)) is higher than in Net 7 by around 50 msec for high frequency of Slow_Read then it decreases for a difference of 15 msec for low frequency of Slow_Read. It should be noted that Fast_Read_1 of nodes 1 to 12 are converted to Fast_Read_0, therefore the average response time of reads (Figure 31(a)) is dominated by the response time of Fast_Read_0.

  The time at which updates reach replicas for Net 8 is higher than Net 7 as shown in Figure 31(f). It should be noticed that updates in Net 8 need to cross less number of hops (wan links) than Net 7 to propagate updates to other replicas. However, the increase in the commit time is higher than the reduction in the communication time. The age returned by Fast_Read is more or less the same for both networks with a slight increase in Net 8. Fast_Read_1 have better age in Net 8 than in Net 7 since all Fast_Read_1 are granted from

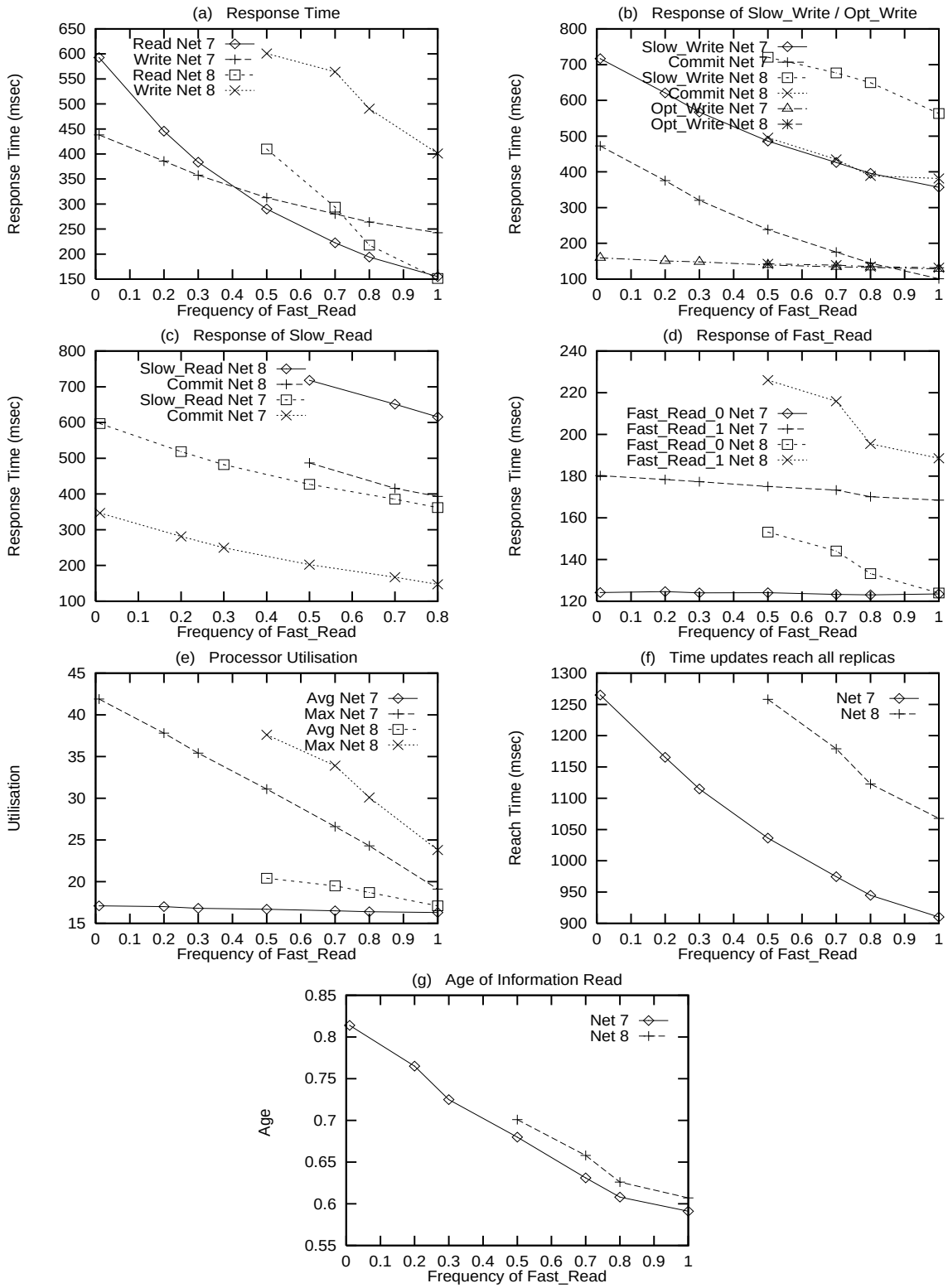


Figure 31: Varying read mix, comparing Net 7 and Net 8

the most up to date copies. However, Fast_Read_0 of nodes 13 to 39 (70% of nodes) have higher age due to the higher reach time. These two factors balance each other and results in an age close to that of Net 7.

In conclusion, it is observed that the performance of Net 8 is much worse than Net 7 in every single metric, except response of Opt_Write and the age which are nearly the same. Further, it could not cope with high loads especially with high frequency of Slow_Write and Slow_Read. This is due to that Net 8 places more load on the resources due to larger quorum size which increases the commit time and the rate of deadlock.

It should be noticed that since Net 8 involves 66 wan_links while Net 7 involves only 48, assuming that lines of Net 8 have the same bandwidth as Net 7 is unfair. A fair comparison would be if we assign the lines of Net 8 a bandwidth less than those of Net 7 (such that the total bandwidth would be constant). In that case the *wan_del* would be 13.75 for Net 8 versus 10 in Net 7. Carrying out such experiments, the extra delay on the lines has caused more increase in the commit time and we failed to reach a steady state for any frequencies of $slow_r$.

### 4.5.3 Comparing Net 7 and Net 9

From Figure 32(b), it is observed that the response of Slow_Write is better in Net 7 by around 50 msec. We notice that the commit time of Slow_Write in Net 9 is higher than in Net 7 by nearly the same amount. The reason is because Net 9 places more load on nodes in the top level as shown in Figure 32(e). This load consists of: more communication overhead due to the extra links attached to these nodes, more propagation overhead and more processing overhead of Fast_Read_1, especially when $fast_{r1}$ increases. This increases the load on the nodes in the top cluster and consequently it increases the processing time and the commit time of Slow_Write. We observe that Net 9 consists of two levels only versus three in Net 7 which is supposed to decrease the communication time, especially of sending and receiving results. However, due to the load placed on nodes in top cluster, the increase in the commit time is higher than the reduction in the communication time. The same comments apply to the response of Slow_Read as shown in Figure 32(c).

The response of Fast_Read_0 and Opt_Write are pretty much the same for both networks. As shown in Figure 32(d), the response of Fast_Read_1 in Net 9 is higher than Net 7 by around 30 msec for low $fast_r$ and the difference increases to 35 msec for high $fast_r$. The reason is, as shown in Figure 32(e), Fast_Read_1 places more load on nodes in level 1 than Net 7. The difference increases as $fast_{r1}$ increases since in Net 9 all Fast_Read_1 are granted from those nodes while in Net 7 the processing of Fast_Read_1 is distributed between nodes in level 1 and 2 which balances the load among the nodes.

From the above, we observe that every operation performance is either the same or better in Net 7 than Net 9 which explains why the average response time of reads and writes (Figure 32(a)) is also better.

As shown in Figure 32(f), the reach time of Net 9 is higher than in Net 7. Although Net 9 consists of 2 levels which should cut down the communication time for updates to reach replicas, however since the commit time of Opt_Write and Slow_Write is higher in Net 9, this delays the propagation of updates and the reach time ends up by being higher. We observe that the difference gets bigger for high $fast_{r1}$. Consequently, the age in Net 9 is slightly higher than Net 7 as shown in Figure 32(g), and the difference increases as $fast_r$ increases.

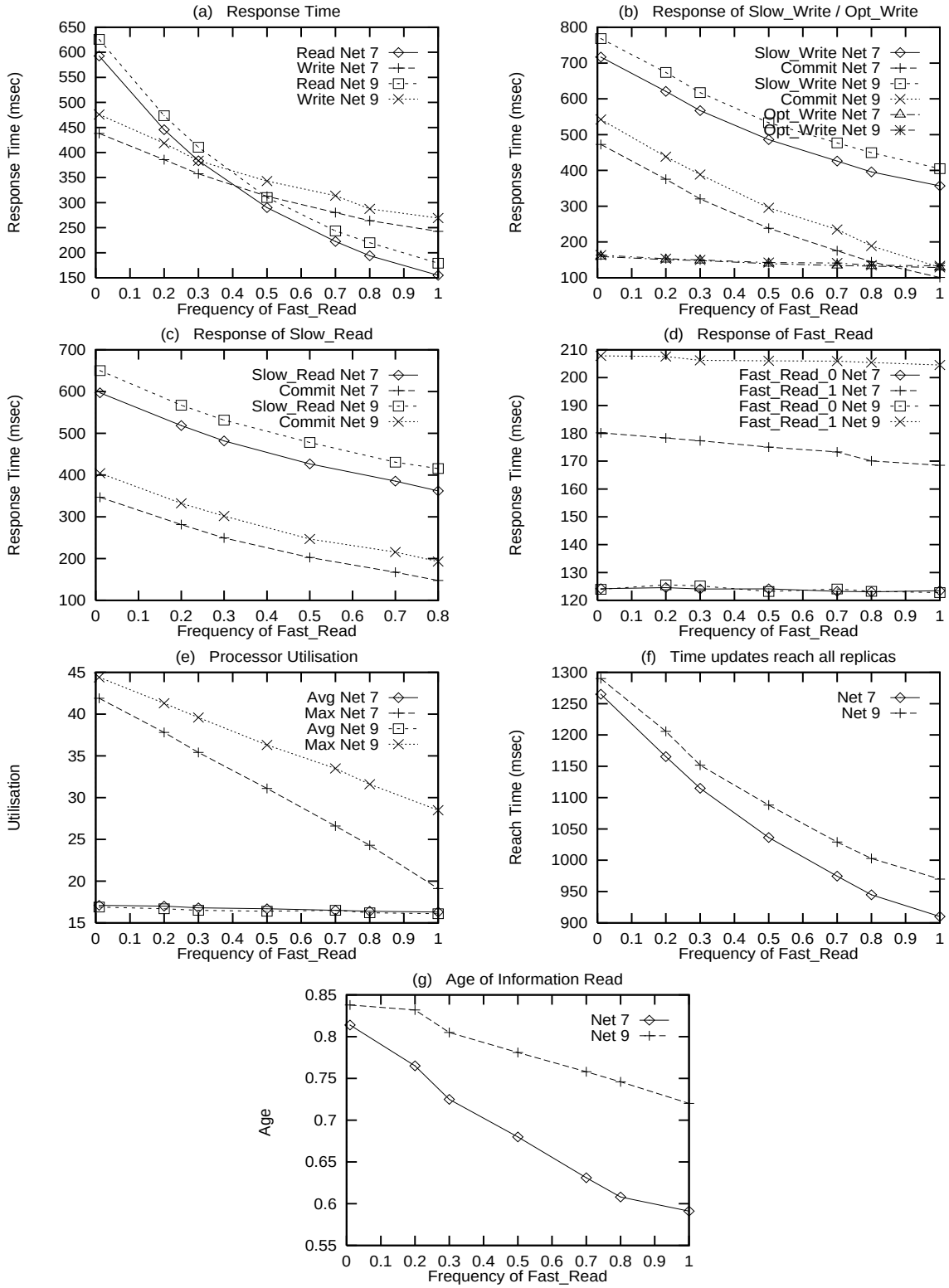


Figure 32: Varying read mix, Comparing Net 7 and Net 9

In conclusion, Net 7 performs better than Net 9 since it distributes the load more evenly among the nodes and does not overload nodes in the top cluster.

## 4.6 Networks of 39 replicas, updates are Fast_Write

These experiments explore the performance over the 39-nodes networks shown in Figure 30 when the load consists of a mix of fast reads and all updates are Fast_Write. The parameter setting is as follows:

| $slow_w$ | $opt_w$ | $fast_w$ | $fast_{r0}$ | $fast_{r1}$ | $slow_r$ | $inter_arr_rate$ | $freq_r$ | $cpu_msg$ | $wan_del$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0-1 | 0-1 | 0 | 1500 | 0.7 | 5 | 30 |

### 4.6.1 Comparing Net 7 and Net 8

Since Net 8 involves 66 wan_links while Net 7 involves only 48, the bandwidth is reassigned per line such that the total bandwidth is constant for fairness of comparison. Therefore, *wan_del* is set to 42 msec for Net 8.

The behaviour of both networks is the same regarding their reaction to the increase in $fast_{r1}$: the response time of Fast_Write and Fast_Read_0 is nearly the same for both networks at low $fast_{r1}$ then it increases slightly as more Fast_Read_1 are introduced, as shown in Figure 33(a) and (b). This is due to that Fast_Read_1 places more load on nodes in the top cluster which increase their utilisation (as shown in Figure 33(c)) and therefore create more contention. The response of Fast_Read_1 increases as $fast_{r1}$ increases due to the extra load on processing resources and communication links. Consequently, the average response time increases as shown in Figure 33(a). The time at which updates reach replicas increases with the increase in $fast_{r1}$ (Figure 33(d)). The age (Figure 33(e)) starts to decrease as Fast_Read_1 are introduced. The decrease persists for low frequency of Fast_Read_1; since Fast_Read_1 have lower age than Fast_Read_0. Then, at $fast_{r1} > 0.5$, the age starts to increase: as more Fast_Read_1 are present, more contention is placed on the resources which increases the time by which updates reach other replicas and deteriorates the age. From here we can come to the same conclusion: an even mix of Fast_Read_0 and Fast_Read_1 is a good mix that compromises losses and benefits.

Comparing Net 7 and Net 8, we observe that Net 8 behaves constantly worse than Net 7. The responses of Fast_Read_0 and Fast_Write are slightly higher in Net 8 for high $fast_{r1}$ (around 10 msec higher). The response of Fast_Read_1 on Net 8 is nearly 50 msec higher than on Net 7. There are two reasons for this increase. First, Net 8 puts more communication overhead on nodes in the top cluster due to the extra links which increase their utilisation and contention. As shown in Figure 33(c), the maximum utilisation is higher in Net 8 by at least 7%. Second, since the delay per link is higher than Net 7, then requests spend more time in communication. Although the traffic per link might be less on certain links on Net 8 than on Net 7, the links are saturated quicker on Net 8 since they have a lower service rate. These two reasons also cause the reach time of Net 8 to be higher by around 60-80 msec, as shown in Figure 33(d), and the age is higher by around 0.4 updates as illustrated in Figure 33(e).

### 4.6.2 Comparing Net 7 and Net 9

As observed from Figure 34, the performance of Net 9 is much worse than Net 7 in every single metric. The average response time of reads and writes are similar for low $fast_{r1}$, then it increases

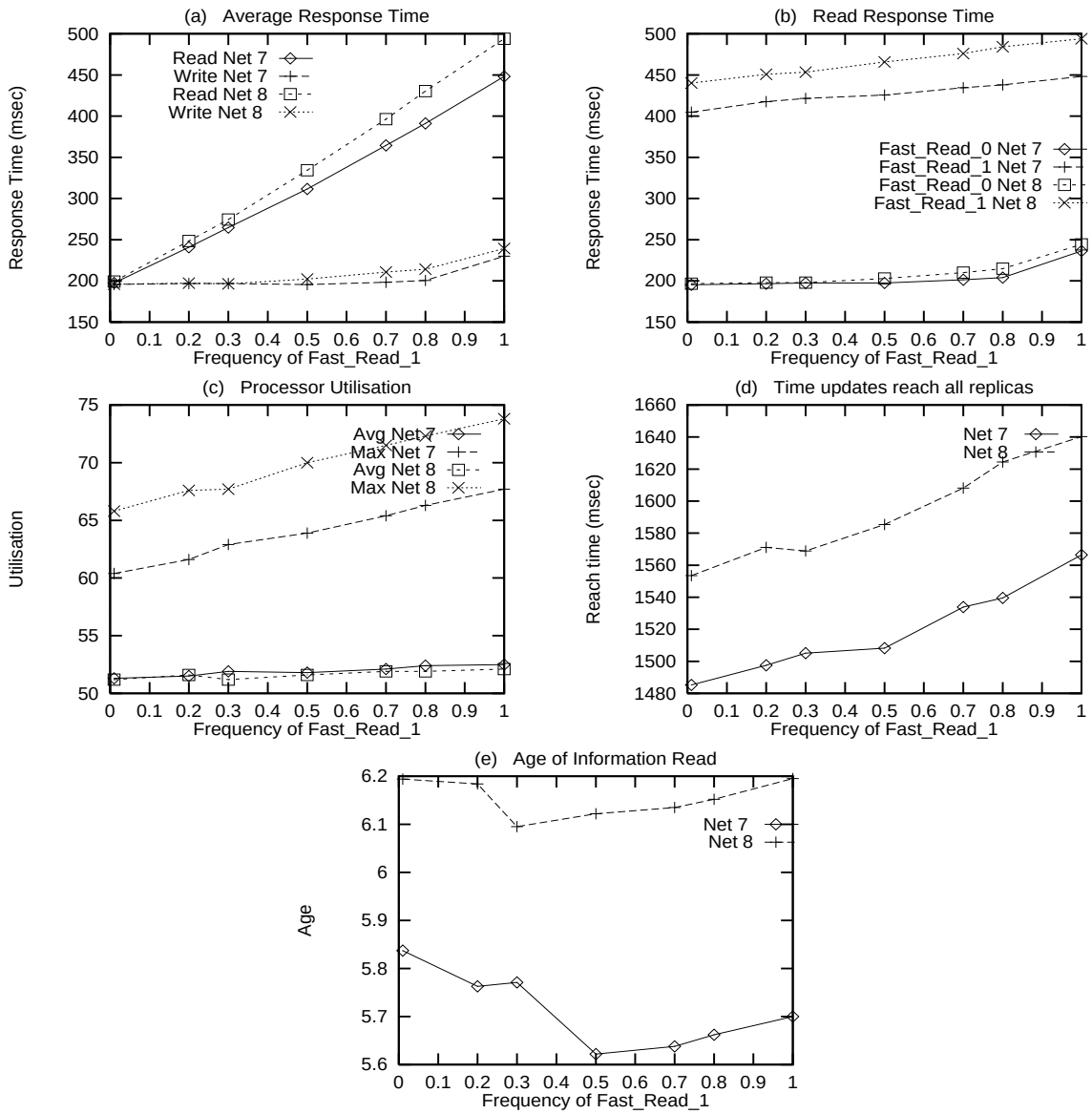


Figure 33: Varying $fast_{r1}$, updates are *Fast_Write*, Comparing Net 7 and Net 8

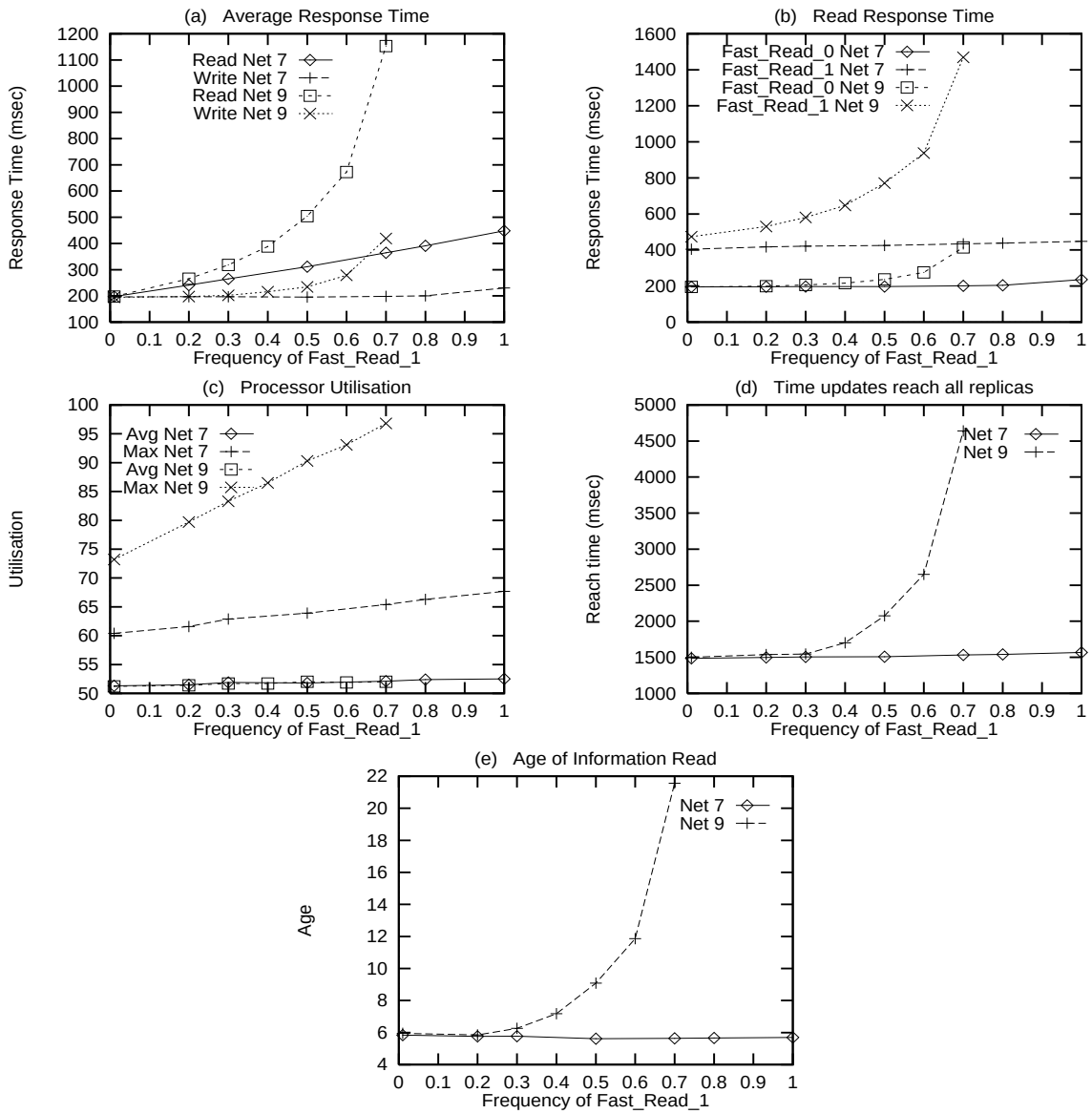


Figure 34: Varying $fast_{r1}$, updates are *Fast_Write*, Comparing Net 7 and Net 9

exponentially for Net 9. The response of Fast_Read_1 is higher in Net 9 even for low $fast_{r1}$, then the increase is steeper as more Fast_Read_1 are introduced. Similarly, the reach time and the age are quite similar for low $fast_{r1}$ then it increases exponentially for Net 9. It is noticed that a steady state could not be reached for Net 9 with $fast_{r1}$ higher than 0.7. The reasons for this behaviour are:

1. Net 9 puts more load on nodes in the top cluster: communication overhead, propagation overhead as well as processing all the Fast_Read_1 requests. As seen in Figure 34(c), the maximum load placed on the processors in Net 9, which is the load on nodes in the top cluster, is much higher than those of Net 7 even for low $fast_{r1}$. The difference gets bigger for high $fast_{r1}$, since all Fast_Read_1 are processed at those nodes (level 1) while in Net 7 they are split between nodes of level 1 and 2. Therefore, in Net 9 level 1 is involved with more communication and processing overhead and the system becomes saturated at $fast_{r1}$=0.7.

2. The load among the communication links are more evenly distributed on Net 7 than in Net 9. For instance, the link between nodes 1 and 4 in Net 9 has to carry for nodes 4,5 and 6 all updates generated from 36 nodes. While in Net 7, it carries for them updates generated from 27 nodes only. This creates more congestion on links in Net 9 and requests spend more time in communication.

It is even noticed that the reach time increases so sharply that we do not see an improvement in the age when introducing Fast_Read_1 in Net 9, even for low $fast_{r1}$. Although in Net 9 the maximum path would consist of 5 wan links while in Net 7 it is 7 wan links, the contention on the resources increases the time spent by a request and we do not see the benefit of having only two levels.

From here we can conclude that Net 9 is worse than Net 7 for every performance metric. In general, we should consider the load placed on nodes in the top cluster and try to spread it to lower levels if possible.

# 5 Verification of the Simulator

In this section, an approximate analytical solution to the model is presented for verifying the results of the simulator.

## 5.1 The analytical model

The system can be modelled as an *Open Queueing Network Model with Multiple Job Classes*. The solution adopted here follows the *Mean Value Analysis* (or *Operational Analysis*) approach described by Lazowska in [LZGS84]. The approach is based on the idea of *observations* where inputs and outputs are average values rather than a probabilistic description of the system behaviour. The technique allows computation of performance measures such as utilisation, throughput and response time by analysing the long term time average behaviour of the system. This technique was chosen as it is efficient, less complex and much faster than the exact solution algorithms -such as mathematical techniques to analyse queueing theory [GH85]- and yet yields sufficiently accurate results [Que82].

The model considers *Separable Queueing Networks* which are a subset of the general networks of queues. In separable queueing networks models, each resource can be separated from the network and its solution evaluated in isolation. The solution of the entire network can then be formed by combining these separate solutions. In an intuitive sense, a separable network has the property that each resource acts (largely) independently of the others.

**Definition of input parameters** used to describe the system :

$C$ = Number of customer classes
$\lambda_c$ = Arrival rate of class $c$ jobs
$K$ = Number of resources
$V_{c,k}$= Number of visits a job of class $c$ performs at resource $k$
$S_{c,k}$= Service requirement per visit of class $c$ job at resource $k$
$D_{c,k}$= Service demand of job class $c$ at resource $k = V_{c,k} * S_{c,k}$

**Equations of the output measures**

While deriving the solution of queueing network model with multiple job classes using FCFS discipline, it is necessary to impose the restriction that all customer classes have the same requirement at each visit to the resource ($S_{c,k}$). The same restriction is imposed by all solutions based on queueing theory [BCMP75, Kel75, Kel76]. It is possible though for different customer classes to require different total number of visits to the resource $V_{c,k}$ thus providing for distinct service demands $D_{c,k}$.

- From the utilisation law
  $U_{c,k}$= Class $c$ utilisation of resource $k = \lambda_k * V_{c,k} * S_{c,k} = \lambda_c * D_{c,k}$
  $U_k$ = Utilisation of resource $k = \sum_{i=1}^{C} U_{i,k}$

- The residence time:
  $R_{c,k}$ = average residence time of class $c$ at resource $k = \frac{D_{c,k}}{1-U_k}$

The solution technique used is based on the assumption that for separable queueing network models the mean queue length seen upon arrival at resource $k$, $A_k(\lambda)$, is equal to the time averaged queue length $Q_k(\lambda)$. The derivation of the above equations can be found in [LZGS84].

In our model, $S_{c,k}$ is not the same for all classes. In order to be able to apply the above solution, we can think of $S^*_{c,k} = 1$ and $V^*_{c,k} = S_{c,k} * V_{c,k}$ while deriving the solution. However, since $D_{c,k} = S_{c,k} * V_{c,k} = S^*_{c,k} * V^*_{c,k}$ and since in the final equations, as shown, the term $D_{c,k}$ is only used, then the equations need not to be changed. For clarity of presentation, the model will be defined in terms of the unequal $S_{c,k}$. Due to the approximation used, a deviation in the results will occur which will be discussed later.

### 5.1.1 Definition of job classes

In order to apply the above solution to our model, we classify jobs in terms of the type of the operation (Fast_Read_0, Fast_Read_1, Fast_Write, Slow_Read, Slow_Write, or Opt_Write) and within one type of operation, it is divided into several classes according to the different phases it passes through e.g. processing, communication, overhead of propagation algorithm and so on. In naming job classes, we use the abbreviation *F0* for Fast_Read_0, *F1* for Fast_Read_1, *FW*

for Fast_Write, $SW$ for Slow_Write and $OW$ for Opt_Write. The different job classes and their descriptions are given in Table 2.

| *Class* | Description |
|---|---|
| $\alpha$*_proc* | class of operation $\alpha$ in the processing phase, where $\alpha$= *F0,F1,FW,SR,SW* or *OW* |
| $\alpha$*_comm* | class of operation $\alpha$ in the processing communication phase, where $\alpha$= *F1,FW,SR,SW* or *OW* |
| $\alpha$*_transc* | class of operation $\alpha$ in the communication phase at transceivers, where $\alpha$= *F1,FW,SR,SW* or *OW* |
| $\alpha$*_const_propg_ovhd* | class of operation $\alpha$ in propagation overhead (constant part), where $\alpha$= *FW,SW* or *OW* |
| $\alpha$*_var_propg_ovhd* | class of operation $\alpha$ in propagation overhead (variable part), where $\alpha$= *FW,SW* or *OW* |
| $\alpha$*_const_2pc_ovhd* | class of operation $\alpha$ in 2PC overhead (constant part), where $\alpha$= *SR,SW* or *OW* |
| $\alpha$*_var_2pc_ovhd* | class of operation $\alpha$ in 2PC overhead (variable part), where $\alpha$= *SR,SW* or *OW* |
| $\alpha$*_lock* | class of operation $\alpha$ in processing (requesting/releasing) locks, where $\alpha$= *SR, SW* or *OW* |
| *OW_undo* | class of Opt_Write operation while undoing an aborted request |
| *OW_abort* | class of Opt_Write operation while executing the abort protocol |

Table 2: Definition of different job classes

### 5.1.2 Definition of the resources

The model considers the following resources:

- Processing resources: $cpu_i \qquad i = 1 : n$
- IO resources: $IO_i \qquad i = 1 : n$
- Transceivers: $transc_i \qquad i = 1 : n_tr$

### 5.1.3 Definition of the Service Times ($S_{c,k}$)

• $S_{\alpha_proc,cpu_k}$ = service time per visit of any operation type in the processing phase at resource $k$

$$= \frac{cpu_req}{power_cpu_k}$$

$$S_{\alpha_proc,IO_k} = \frac{io_req}{power_io_k}$$

• $S_{\alpha_comm,cpu_k}$ = service time per visit of any operation type at $cpu_k$ for communication processing

$$= \frac{cpu_msg}{power_cpu_k}$$

• $S_{\alpha_transc,transc_k}$ = service time per visit at transceiver $k$ by any type of operation = $\frac{wan_del}{power_transc_k}$

• $S_{\alpha_const_propg_ovhd,cpu_k}$ = service time per visit at $cpu_k$ by any write operation while performing the constant part of the propagation overhead = $\frac{prop_ovhd_const}{power_cpu_k}$

• $S_{\alpha_var_propg_ovhd,cpu_k}$ = service time per visit at $cpu_k$ by any write operation while performing

the variable part of the propagation overhead $= \frac{prop_ovhd_var}{power_cpu_k}$
• $S_{\alpha_const_2pc_ovhd,cpu_k}$ = service time per visit at $cpu_k$ by an operation $\alpha$ while performing the constant part of the 2PC overhead, where $\alpha$= $SR,SW,OW = \frac{pc_ovhd_const}{power_cpu_k}$
• $S_{\alpha_var_2pc_ovhd,cpu_k}$ = service time per visit at $cpu_k$ by an operation $\alpha$ while performing the variable part of the 2PC overhead, where $\alpha$= $SR,SW,OW = \frac{pc_ovhd_var}{power_cpu_k}$
• $S_{\alpha_lock,cpu_k}$ = service time per visit at $cpu_k$ by an operation $\alpha$ while requesting or releasing a lock, where $\alpha$= $SR,SW,OW = \frac{cpu_lock}{power_cpu_k}$
• $S_{OW_undo,cpu_k}$ = service time per visit of an Opt_Write while undoing an aborted write at resource $k$

$$= \frac{cpu_undo}{power_cpu_k}$$

$$S_{OW_undo,IO_k} = \frac{io_undo}{power_io_k}$$

• $S_{OW_abort,cpu_k}$ = service time per visit of an Opt_Write while executing the abort protocol overhead at resource $k = \frac{cpu_abort}{power_cpu_k}$

### 5.1.4 Definition of arrival rates λs

The arrival rates of different job classes depend on the type of the operation. Assuming $\lambda = 1/inter_arr_rate$ we have:

• $\lambda_{F0_*} = \lambda * freq_r * fast_{r0}$
• $\lambda_{F1_*} = \lambda * freq_r * fast_{r1}$
• $\lambda_{FW_*} = \lambda * (1 - freq_r) * fast_w$
• $\lambda_{SR_*} = \lambda * freq_r * slow_r$
• $\lambda_{SW_*} = \lambda * (1 - freq_r) * slow_w$
• $\lambda_{OW_*} = \lambda * (1 - freq_r) * opt_w$

### 5.1.5 Notations and Definitions

In the following, we introduce some notations and definitions that will be used in the presentation of the solution.

• $Nb_i$ = {$j$: node $j$ is a neighbour of node $i$ } = set of neighbours of $i$
• $Ch_i$ = {$j$: node $j$ is a child of node $i$ } = set of children of $i$
• $C_i$ = Cluster id which node $i$ belongs to
• $i \leftrightarrow j$: nodes $i$ and $j$ are directly connected
• $i \not\leftrightarrow j$: nodes $i$ and $j$ are *not* directly connected
• $P_i$ = id of the parent node of $i$ (=0 if $i$ is in the top cluster)
• $n_child_i$ = number of children of node $i$ = $\mid Ch_i \mid$
• $n_neigh_i$ = number of neighbours of node $i$ = $\mid Nb_i \mid$
• $n_par_i$ = number of parents of node $i$

$$= \begin{cases} 1 & \text{if } P_i \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

- $descendant_i$ = number of descendant nodes from node $i$

$$= \begin{cases} 0 & \text{if } \mid Ch_i \mid = 0 \\ \sum_{s,\forall s \in Ch_i}(1 + descendant_s) & \text{otherwise} \end{cases}$$

- $direct_neigh_i$ = number of neighbours of $i$ that are directly connected to it
  $= \mid \{j : j \in Nb_i \wedge j \leftrightarrow i\} \mid$
- $direct_child_i$ = number of children of $i$ that are directly connected to it
  $= \mid \{j : j \in Ch_i \wedge j \leftrightarrow i\} \mid$
- $direct_par_i$ = number of parents of $i$ that are directly connected to it

$$= \begin{cases} 1 & \text{if } P_i \neq 0 \wedge P_i \leftrightarrow i \\ 0 & \text{otherwise} \end{cases}$$

- $msg_fm_P_i$ = total number of messages received by $i$ from its parent through propagation
  $= n - descendant_{P_i} + \sum_{s,\forall s \in Ch_{P_i} \wedge C_s \neq C_i}(1 + descendant_s)$

Node $i$ receives from its parent messages generated from all nodes except the ones generated from the descendants of the parent. The third term accounts for the children of $P_i$, and their descendants, which are in a different cluster than $i$. Node $i$ receives messages generated from those nodes via $P_i$.

- $Path_{xy}$ = set of nodes that are on the the path from node $x$ to node $y$
- $n_root$ = number of nodes in the top cluster
- $q_size$ = the quorum size = $\lceil \frac{n_root}{2} \rceil$
- $pr_co_i$ = probability that node $i$ is a coordinator

$$= \begin{cases} \frac{1+descendant_i}{n} & \text{if } P_i = 0 \\ 0 & \text{otherwise} \end{cases}$$

- $pr_quo_i \mid_{co \neq i}$ = probability that node $i$ belongs to the quorum given that it is not a CO

$$= \begin{cases} \frac{q_size-1}{n_root-1} & \text{if } P_i = 0 \\ 0 & \text{otherwise} \end{cases}$$

- $pr_quo_i$ = probability that a node $i$ belongs to the quorum

$$= \begin{cases} (pr_co_i + \frac{n-(descendant_i+1)}{n} * pr_quo_i \mid_{co \neq i} & \text{if } P_i = 0 \\ 0 & \text{otherwise} \end{cases}$$

Node $i$ is in the quorum if it is a CO or if it is selected by other nodes (other than itself and its descendants) to be a quorum node. The later case happens with probability $pr_quo_i \mid_{co \neq i}$.

### 5.1.6 Computation of the utilisation of the resources

In order to compute the utilisation of the resources, the total number of visits paid at each resource needs to be computed. We will proceed by analysing the number of visits that each type of operation performs at the different resources ($V_{c,k}$).

**Number of visits by** Fast_Read_0

Since each node processes its own requests, then
$V_{F0_proc,cpu_i} = V_{F0_proc,IO_i} = 1 \qquad \forall i = 1:n$

**Number of visits by** Fast_Read_1

$\star$ Resources of node $i$ process Fast_Read_1 requests of its children. Further, since it is assumed that for nodes in the top cluster, Fast_Read_1 are converted to Fast_Read_0, then node $i$ processes its own requests in case $i$ is in the top cluster. Therefore,
$V_{F1_proc,cpu_i} = V_{F1_proc,IO_i} = n_child_i + (1 - n_par_i) \qquad \forall i = 1:n$

$\star$ $V_{F1_comm,cpu_i}$ = number of communication messages processed by $i$ for a Fast_Read_1 request
$= (2 * n_par_i) + (2 * n_child_i) + (2* \mid ND_i \mid)$
where $ND_i = \{j : j \in Nb_i \wedge j \leftrightarrow i \wedge j \not\leftrightarrow P_j \wedge i \leftrightarrow P_i \}$
The first term considers the communication messages resulting from $i$'s own requests (to send requests and receive results) in case it is not in the top cluster, while the second term accounts for the communication messages sent and received from children of $i$ to grant their requests. The third term accounts for the communication messages $i$ processes while acting as an intermediate node between its neighbours that are not directly connected to their parent while $i$ is.

$\star$ Computing the number of visits to transceiver $k$, connecting nodes $i$ to $j$ (or node $j$ to $i$)

- if $P_j = i \qquad V_{F1_transc,transc_k} = 1+ \mid ND_j \mid$
  If $i$ is the parent of $j$, then transceiver $k$ processes requests (or results) of node $j$ and requests (or results) of neighbours of $j$ which are directly connected to $j$ but not to their parent.

- if $i$ and $j$ are neighbours and they are not in the top cluster

$$V_{F1_transc,transc_k} = \begin{cases} 1 & \text{if } i \leftrightarrow P_i \wedge j \not\leftrightarrow P_j \\ 0 & \text{otherwise} \end{cases}$$

  If $j$ is not directly connected to its parent but $i$ is, then transceiver $k$ carries the requests (or results) of node $j$.

**Number of visits by** Fast_Write

$\star$ Since each node processes the Fast_Write generated by all other nodes, then the total number of visits at resource $i$ for processing is given by:
$V_{FW_proc,cpu_i} = V_{FW_proc,IO_i} = n \qquad \forall i = 1:n$

$\star$ $V_{FW_comm,cpu_i}$ = number of communication messages processed by node $i$ for propagating Fast_Write
$= (direct_neigh_i + direct_child_i + direct_par_i)$

$$
\begin{aligned}
&+ (msg_fm_P_i * (1 + direct_child_i)) \\
&+ (\textstyle\sum_{s,\forall s \in Ch_i}(1 + descendant_s)(1 + direct_neigh_i + direct_par_i + \mid DC_{i,s} \mid)) \\
&+ ((1 + direct_child_i) * \textstyle\sum_{w,\forall w \in Nb_i}(1 + descendant_w)) \\
&+ (msg_fm_P_i * \mid ND_i \mid) \\
&+ (\textstyle\sum_{w,\forall w \in Nb_i \wedge i \leftrightarrow w \wedge i \leftrightarrow P_i \wedge w \not\leftrightarrow P_i}(1 + descendant_w)) \\
&+ (\textstyle\sum_{w,\forall w \in Nb_i} \sum_{y,\forall y \in Nb_i \wedge y \leftrightarrow i \wedge w \neq y \wedge i \in Path_{wy}}(1 + descendant_w) + (1 + descendant_y))
\end{aligned}
$$

where $DC_{i,s} = \{\ s1 : s1 \in Ch_i \wedge C_{s1} \neq C_s \wedge i \leftrightarrow s1\ \}$

Term 1: since a message $i$ originates is sent to its neighbours, parent and children, then a communication message is sent to each of them that is directly connected to $i$.

Term 2: For every message received from the parent, it involves one communication message for its receipt. Further, since it is sent to all children, then a communication message is sent to each child that is directly connected to $i$.

Term 3: For every message received from a child (which involves the child itself and its descendants), a communication message is needed for its receipt. Further, such a message is sent to: its neighbours, parent and children belonging to other clusters than the one the message is coming from. Then, a communication message is needed for each of those nodes that is directly connected to $i$.

Term 4: For every message received from a neighbour (which involves the neighbour itself and its descendants); a communication message is needed for the receipt of the message. Further, since the message is sent to all children of $i$, then a communication message is sent to each of them that is directly connected to $i$.

Term 5 and 6: For each neighbour $w$ that is directly connected to $i$ but not to the parent, $i$ acts as an intermediate node between $w$ and the parent; provided that $i$ is directly connected to the parent. For each such a neighbour $w$, $i$ processes communication messages sent from the parent to $w$ (term 5) and messages originating from neighbour $w$ and its descendants to the parent (term 6).

Term 7: For a neighbour $y$ that is not directly connected to another neighbour $w$, $i$ acts as intermediate node between $w$ and $y$ provided that $i$ is directly connected to both of them. For each such a neighbour $y$, $i$ needs to process a communication message for each message $y$ or its descendants originates before passing it to $w$ and vice versa.

$\star$ Since the propagation algorithm runs for every update generated at any node, then

$V_{FW_const_propg_ovhd,cpu_i} = n$

$\star$ During propagation, a Fast_Write that $i$ originates is sent to neighbours, parent and children. For messages $i$ receives from the parent, they are sent to children of $i$. For messages originating at a neighbour or its descendants, $i$ receives them from the neighbour and are sent to children of $i$. Finally, messages originating from a child $s$ or its descendants are sent to neighbours, parent and children of other cluster than $C_s$. Therefore, the number of visits that Fast_Write pays to $cpu_i$ for performing the variable part of the propagation overhead is given by:

$$
\begin{aligned}
V_{FW_var_propg_ovhd,cpu_i} = &(n_neigh_i + n_child_i + n_par_i) \\
&+ (msg_fm_P_i * n_child_i) \\
&+ (n_child_i * \textstyle\sum_{w,\forall w \in Nb_i}(1 + descendant_w)) \\
&+ (\textstyle\sum_{s,\forall s \in Ch_i}(1 + descendant_s) * (n_neigh_i + n_par_i + \mid SC_{i,s} \mid))
\end{aligned}
$$

where $SC_{i,s}$ = set of children of $i$ that are not in the same cluster as $s$
$= \{s1 : s1 \in Ch_i \wedge C_{s1} \neq C_s\ \}$

$\star$ Computing the number of visits to transceiver $k$, connecting nodes $i$ to $j$

- if $P_j = i$

  $V_{FW_transc,transc_k} = 1 + msg_fm_P_i + \sum_{w,\forall w \in Nb_i}(1 + descendant_w) + \sum_{s,\forall s \in Ch_i \wedge C_s \neq C_j}(1 + descendant_s)$

  If $i$ is the parent of $j$, then transceiver $k$ processes messages $i$ originates, messages received from the parent of $i$, messages originating at neighbours of $i$ and their descendants and finally, messages originating at children of $i$ which do not belong to the same cluster as that of $j$ and their descendants.

- if $i \in Ch_j$

  $V_{FW_transc,transc_k} = 1+\sum_{s,\forall s \in Ch_i}(1+descendant_s)+\sum_{w,\forall w \in Nb_i \wedge i \leftrightarrow w \wedge w \not\leftrightarrow j}(1+descendant_w)$

  If $i$ is the child of $j$ then, transceiver $k$ processes messages $i$ originates, messages originating at children of $i$ and their descendants and messages originating at neighbours of $i$ - not directly connected to their parent ($j$) but that are directly connected to $i$ - and their descendants.

- if $i \in Nb_j$

  $V_{FW_transc,transc_k} = 1+\sum_{s,\forall s \in Ch_i}(1+descendant_s)+\sum_{w,\forall w \in Nb_i \wedge i \leftrightarrow w \wedge w \not\leftrightarrow j}(1+descendant_w)$

  If $i \leftrightarrow P_i \wedge j \not\leftrightarrow P_i$ then $V_{FW_transc,transc_k} += msg_fm_P_i$

  If $i$ is a neighbour of $j$, then transceiver $k$ processes messages $i$ originates, messages originating at children of $i$ and their descendants, messages originating at neighbours of $i$ - not directly connected to their neighbour $j$ but that are directly connected to $i$ - and their descendants, and messages received from the parent if $i$ is directly connected to the parent while $j$ is not.

**Number of visits by** Slow_Write

$\star$ Since each node processes the Slow_Write generated by all other nodes, then the total number of visits at resource $i$ for processing is given by:

$V_{SW_proc,cpu_i} = V_{SW_proc,IO_i} = n \qquad \forall i = 1 : n$

$\star$ Node $i$ acting as a CO executes the overhead of 2PC twice; once for each phase. Therefore,

$V_{SW_const_2pc_ovhd,cpu_i} = pr_co_i * 2 * n$

$\star$ Node $i$ acting as a CO executes the variable part of 2PC for each node participating in the quorum twice; once for each phase. Therefore,

$V_{SW_var_2pc_ovhd,cpu_i} = pr_co_i * 2 * n * q_size$

$\star$ Node $i$ participating in the quorum requests and releases locks for each update. Therefore,

$V_{SW_lock,cpu_i} = pr_quo_i * 2 * n$

⋆ Since the propagation algorithm runs for every update generated at any node, then
$V_{SW_const_propg_ovhd,cpu_i} = n$

⋆ For node $i$ executing the variable part of the propagation overhead:

- if $P_i \neq 0$ (Node $i$ is not in the top cluster)
  For updates originating at one of its children, $i$ propagates the messages to its children except the one the message originated from, since that child receives a commit. For any other update, $i$ propagates the message to all children. Therefore,
  $V_{SW_var_propg_ovhd,cpu_i} = n_child_i(n_child_i - 1) + (n - n_child_i)n_child_i$
- if $P_i = 0$ (Node $i$ is in the top cluster)
  $V_{SW_var_propg_ovhd,cpu_i} = ((descendant_i - n_child_i)(n_child_i + n_neigh_i - (q_size - 1)))$
  $+ (n_child_i(n_child_i - 1 + n_neigh_i - (q_size - 1)))$
  $+ (n_child_i + n_neigh_i - (q_size - 1))$
  $+ ((n - (descendant_i + 1))n_child_i)$

  Node $i$ being a CO for its descendants (excluding children), propagates the message to its children and to neighbours not belonging to the quorum (term 1). Node $i$ being a CO for its children, propagates the message to its children, but not to the one the message came from, and to neighbours not in quorum (term 2). Node $i$ being CO for itself, propagates the message to its children and neighbours not in quorum (term 3). Finally, node $i$ not acting as a CO propagates the message to its children only.

⋆ $V_{SW_comm,cpu_i} = (2 * n_par_i)$
$+ (2 * (descendant_i + \sum_{\forall j \in Nb_i \wedge j \not\leftrightarrow P_j \wedge j \leftrightarrow i \wedge i \leftrightarrow P_i}(1 + descendant_j)) * n_par_i)$
$+ (2 * (1 - n_par_i) * descendant_i)$
$+ (pr_co_i * n * 3 * (q_size - 1))$
$+ ((pr_quo_i - pr_co_i) * n * 3)$
$+ (direct_child_i * n)$
$+ (pr_co_i * n * (n_neigh_i - (q_size - 1)))$
$+ ((1 - pr_quo_i)(n - 1))$
$+ (\mid ND_i \mid * ((\mid ND_i \mid -1) + (n - \mid ND_i \mid)) * n_par_i)$

The first five terms consider the communication messages generated from executing the 2PC. For a node not in the top cluster a communication message is needed to send and receive results from CO (term 1). Node $i$ acts as an intermediate node between its descendants and its parent and between its neighbours (and their descendants) that are not directly connected to the parent for sending and receiving results from CO (term 2). Node $i$ as a CO executes 2 communication messages for each of its descendants: to receive request from origin and to send commit to origin (term 3). Node $i$ as a CO executes 3 communication messages (send vote_req, receive vote_yes and send commit) for every node participating in the quorum (term 4). Finally, $i$ being in the quorum but not a CO executes 3 communication messages (receive vote_req, send vote_yes and receive commit).

The last four terms consider the communication messages generated from the propagation of committed updates. Since a node propagates each message to its children, then a communication message is sent to children that are directly connected to $i$ (term 6). Since CO sends a propagation message to neighbours not in quorum then a communication message is needed for each (term 7). For each node not in quorum, a communication message is needed to receive the propagated message for each update originating at any other node but itself (term 8). Node $i$ acts as an intermediate node between its parent and its neighbours that are not directly connected to the parent (term 9).

$\star$ Computing the number of visits to transceiver $k$, connecting nodes $i$ to $j$:

- if $P_i = P_j = 0$ (Node $i$ and $j$ are in the top cluster)

  When $i$ acts as a CO, it sends two messages to $j$ if $j$ belongs to the quorum: vote_req and commit, and it sends one propagation message to $j$ if $j$ is not in the quorum. Finally, when $j$ acts as a CO, transceiver $k$ processes vote_yes message sent from $i$ to $j$, if $i$ belongs to the quorum. Therefore,

  $$V_{SW_transc,transc_k} = n * (pr_co_i * pr_quo_j \mid_{co \neq j} * 2 + pr_co_i * (1 - pr_quo_j \mid_{co \neq j}) + pr_co_j * pr_quo_i \mid_{co \neq i})$$

- if $i \in Ch_j$

  If $i$ is a child of $j$, then transceiver $k$ will process messages $i$ originates, messages originating at children of $i$ and their descendants and messages originating at neighbours of $i$ - not directly connected to their parent ($j$) but that are directly connected to $i$ - and their descendants. These messages are processed by transceiver $k$ while sent from the origin node to CO.

  $$V_{SW_transc,transc_k} = 1 + \sum_{s,\forall s \in Ch_i}(1 + descendant_s) + \sum_{w,\forall w \in Nb_i \wedge i \leftrightarrow w \wedge w \not\leftrightarrow j}(1 + descendant_w)$$

- if $P_j = i$

  If $i$ is the parent of $j$, then the same number of visits as above are performed for sending the commit from CO to the origin. Also, transceiver $k$ processes every message generated at any node to pass it on for propagation. Therefore,

  $$V_{SW_transc,transc_k} = 1 + \sum_{s,\forall s \in Ch_j}(1 + descendant_s) + \sum_{w,\forall w \in Nb_j \wedge j \leftrightarrow w \wedge w \not\leftrightarrow i}(1 + descendant_w) + n$$

- if $i \in Nb_j \wedge P_i \neq 0 \wedge i \leftrightarrow P_i \wedge j \not\leftrightarrow P_j$

  If $i$ is a neighbour of $j$, both are not in the top cluster, and $i$ is directly connected to the parent while $j$ is not, then transceiver $k$ processes the commit messages sent to $j$ and its descendants for updates they originate. Also, it processes updates generated at any node to pass it on for propagation. Therefore,

  $$V_{SW_transc,transc_k} = 1 + \sum_{s,\forall s \in Ch_j}(1 + descendant_s) + n$$

- if $i \in Nb_j \wedge P_i \neq 0 \wedge j \leftrightarrow P_i \wedge i \not\leftrightarrow P_i$

  If $i$ is a neighbour of $j$, both are not in the top cluster, and $j$ is directly connected to the parent while $i$ is not, then transceiver $k$ processes org_req originating at $i$ and its descendants when they are sent to CO. Therefore,

$V_{SW_transc,transc_k} = 1 + \sum_{s,\forall s \in Ch_i}(1 + descendant_s)$

**Number of visits by** Slow_Read

$\star$ Since a node performs a Slow_Read only if it belongs to the quorum, and since a node is selected from the quorum to execute the request with equal probability $1/q_size$. Then,

$V_{SR_proc,cpu_i} = V_{SR_proc,IO_i} = \frac{pr_quo_i * n}{q_size}$

$\star$ $V_{SR_const_2pc_ovhd,cpu_i} = V_{SW_const_2pc_ovhd,cpu_i}$

$\star$ $V_{SR_var_2pc_ovhd,cpu_i} = V_{SW_var_2pc_ovhd,cpu_i}$

$\star$ $V_{SR_lock,cpu_i} = V_{SW_lock,cpu_i}$

$\star$ $V_{SR_comm,cpu_i} = (2 * n_par_i)$
$+ (2 * (descendant_i + \sum_{\forall j \in Nb_i \wedge j \not\leftrightarrow P_j \wedge j \leftrightarrow i \wedge i \leftrightarrow P_i}(1 + descendant_j)) * n_par_i)$
$+ (2 * (1 - n_par_i) * descendant_i)$
$+ (pr_co_i * n * (3 * (q_size - 1) + \frac{q_size-1}{q_size}))$
$+ ((pr_quo_i - pr_co_i) * n * (3 + \frac{1}{q_size}))$

The first three terms are the same as for Slow_Write. The fourth term considers $i$ being a CO, in which case $i$ processes 3 communication messages (send vote_req, receive vote_yes and send commit) for every node participating in the quorum. Further, in case $i$ is not selected to perform the read (which happens with probability $\frac{(q_size-1)}{q_size}$), $i$ as a CO performs one communication message to receive the results from the selected node. The last term accounts for $i$ being in the quorum but not a CO, so $i$ processes 3 communication messages (receive vote_req, send vote_yes and receive commit). Further, if $i$ is selected to perform the read (which is with probability $1/q_size$), $i$ performs one communication message to send the results to CO.

$\star$ Computing the number of visits to transceiver $k$, connecting nodes $i$ to $j$:

- if $P_i = P_j = 0$ (Node $i$ and $j$ are in the top cluster)
  When $i$ acts as a CO, it sends 2 messages to $j$ if $j$ belongs to the quorum: vote_req and commit. When $j$ acts as a CO, transceiver $k$ processes vote_yes message sent from $i$ to $j$ if $i$ belongs to the quorum. Also, if $i$ is chosen to perform the read (with probability $1/q_size$), it sends one message to $j$ carrying the results. Therefore,
  $V_{SR_transc,transc_k} = n(pr_co_i * pr_quo_j \mid_{co \neq j} * 2 + pr_co_j * pr_quo_i \mid_{co \neq i} * (1 + \frac{1}{q_size}))$
- if $i \in Ch_j$
  $V_{SR_transc,transc_k} = V_{SW_transc,transc_k}$
- if $P_j = i$
  If $i$ is the parent of $j$, then transceiver $k$ will process messages $j$ originates, messages originating at children of $j$ and their descendants and messages originating at neighbours of $j$ - not directly connected to their parent ($i$) but that are directly connected to $j$ - and their descendants. These are the commit messages while sent from CO to the origin node.

$V_{SR_transc,transc_k} = 1+\sum_{s,\forall s \in Ch_j}(1+descendant_s)+\sum_{w,\forall w \in Nb_j \wedge j \leftrightarrow w \wedge w \not\leftrightarrow i}(1+descendant_w)$

- if $i \in Nb_j \wedge P_i \neq 0 \wedge i \leftrightarrow P_i \wedge j \not\leftrightarrow P_j$

  If $i$ is a neighbour of $j$, both are not in the top cluster, and $i$ is directly connected to the parent while $j$ is not, then transceiver $k$ processes the commit messages sent to $j$ and its descendants for requests they originate. Therefore,

  $V_{SR_transc,transc_k} = 1 + \sum_{s,\forall s \in Ch_j}(1 + descendant_s)$

- if $i \in Nb_j \wedge P_i \neq 0 \wedge j \leftrightarrow P_i \wedge i \not\leftrightarrow P_i$

  If $i$ is a neighbour of $j$, both are not in the top cluster, and $j$ is directly connected to the parent while $i$ is not, then transceiver $k$ processes org_req originating at $i$ and its descendants when they are sent to CO. Therefore,

  $V_{SR_transc,transc_k} = 1 + \sum_{s,\forall s \in Ch_i}(1 + descendant_s)$

**Number of visits by** Opt_Write

We will denote by $n_opt_i$ the number of nodes that consider $i$ an optimistic node. Since in our model we consider a node that is on the path from the origin node to CO as an optimistic node, then $n_opt_i$ consists of the node itself, its descendants, its neighbours that are directly connected to it but not to the parent and their descendants. It is given by:

$$n_opt_i = \begin{cases} 0 & \text{if } P_i = 0 \\ 1 + descendant_i + \sum_{j,\forall j \in Nb_i \wedge j \not\leftrightarrow P_j \wedge j \leftrightarrow i \wedge i \leftrightarrow P_i}(1 + descendant_j) & \text{if } P_i \neq 0 \end{cases}$$

$\star$ A node in the top cluster executes every update generated from any node in the top cluster, as these updates always commit. Further, it executes updates generated from other nodes in the hierarchy in case they commit. For a node $i$ not in the top cluster, it executes updates generated at nodes in the top cluster, updates generated at a node that consider $i$ an *optimistic node* and updates generated at the rest of the nodes provided they commit. Therefore,

$V_{OW_proc,cpu_i} = V_{OW_proc,IO_i}$
$= n_root + n_opt_i + (n - n_roots - n_opt_i) * pr_commit$

$\star$ $V_{OW_const_2pc_ovhd,cpu_i}$, $V_{OW_var_2pc_ovhd,cpu_i}$ and $V_{OW_lock,cpu_i}$ are the same as for Slow_Write since 2PC is executed whether the update will commit or abort.

$\star$ A node propagates messages for updates generated at nodes in the top cluster -as they always commit- and for updates generated at other nodes provided they commit. Therefore,

$V_{OW_const_propg_ovhd,cpu_i} = n_root + (n - n_root) * pr_commit$

$\star$ For node $i$ executing the variable part of the propagation overhead:

- if $P_i \neq 0$ (Node $i$ is not in the top cluster)

  For updates generated at a node in the top cluster, $i$ propagates them to all children. For updates generated at a child $j$ or one of its descendants, $i$ propagates the message to its children except the child $j$ itself, since $j$ is either the origin node or an optimistic node and in both cases it receives a commit message rather than a propagation message.

Also, $i$ excludes children that are on the path from $j$ to $i$ as these are optimistic nodes that will receive a commit as well. For updates generated at the rest of the nodes, $i$ sends them to all children. The last two cases are considered provided that the update commits. Therefore,

$$V_{OW_var_2pc_ovhd,cpu_i} = n_root * n_child_i + pr_commit * \\ ((\sum_{\forall j \in Ch_i}(1 + descendants_j)(n_child_i - 1 - \mid Path_{ji} \mid)) + \\ (n - n_root - \sum_{\forall j \in Ch_i}(1 + descendant_j)(n_child_i)))$$

- if $P_i = 0$ (Node $i$ is in the top cluster)

$$V_{OW_var_propg_ovhd,cpu_i} = (\sum_{\forall j \in Ch_i}(1 + descendant_j) * \\ (n_child_i - 1 - \mid Path_{ji} \mid + n_neigh_i - (q_size - 1)) * pr_commit) \\ + (n_child_i + n_neigh_i - (q_size - 1)) \\ + (n_root - 1)(n_child_i) \\ + (n - n_root - descendant_i) * n_child_i * pr_commit$$

The first term considers the case when node $i$ is a CO for a child $j$ or its descendants. Since $j$ (whether it has originated the message or one of its descendants did) already has executed the update, then $i$ propagates it to its children except $j$. Also, $i$ excludes children that are on the path from $j$ to $i$ as these are optimistic nodes that will receive a commit. Further, $i$ sends the message to its neighbours not belonging to the quorum. This term is considered only if the update commits. The second term accounts for updates generated by $i$ (which is a CO for itself), so it sends messages to children and to neighbours that do not belong to the quorum. The third term considers messages generated by neighbours of $i$, which $i$ sends to its children. Finally, the last term accounts for updates generated at any other node which are sent by $i$ to its children provided they commit.

$$\star\ V_{OW_comm,cpu_i} = (2 * n_par_i) \\ + (2 * (descendant_i + \sum_{\forall j \in Nb_i \wedge j \not\leftrightarrow P_j \wedge j \leftrightarrow i \wedge i \leftrightarrow P_i}(1 + descendant_j)) * n_par_i) \\ + (2 * (1 - n_par_i) * descendant_i) \\ + (pr_co_i * n * 3 * (q_size - 1)) \\ + ((pr_quo_i - pr_co_i) * n * 3) \\ + (direct_child_i * (n_root + (n - n_root) * pr_commit)) \\ + (pr_co_i * (n_neigh_i - (q_size - 1)) * (n_root + (n - n_root) * pr_commit)) \\ + ((1 - pr_quo_i)(n_root + (n - n_root - n_opt_i) * pr_commit)) \\ + (n_root * \mid ND_i \mid + (\sum_{\forall w \in Nb_i \wedge i \leftrightarrow w \wedge w \not\leftrightarrow P_i \wedge i \leftrightarrow P_i}(1 + descendant_w) \\ * (\mid ND_i \mid - 1) + (n - \sum_{\forall w \in Nb_i \wedge i \leftrightarrow w \wedge w \not\leftrightarrow P_i \wedge i \leftrightarrow P_i}(1 + descendant_w) \\ - n_root) * \mid ND_i \mid) * pr_commit)$$

The first five terms address messages generated by 2PC and are the same as for Slow_Write. Terms 6 and 7 are the same as for Slow_Write, but they apply only for updates that commit. For each node $i$ not belonging to the quorum, a communication message is needed to receive the propagated update for each Opt_Write originating at any node, excluding itself and nodes that consider $i$ an optimistic node, provided that the update will commit (term 8). The last term considers when node $i$ acts as an intermediate node for a neighbour $w$. For messages originating at $w$ or one of its descendants, then $w$ is an optimistic node; so $i$ does not send $w$ the propagation message. Therefore, it sends it to the remaining intermediate neighbours (provided that

the update commits). For messages originating elsewhere, $i$ sends the propagation message to all these intermediate neighbours provided the update commits.

$\star$ Node $i$ will undo the operation for each aborted update that originated at a node that considered $i$ an optimistic node . Therefore, for nodes not in the top cluster:

$V_{OW_undo,cpu_i} = V_{OW_undo,IO_i} = n_opt_i * (1 - pr_commit)$

$\star$ A node participating in the quorum will execute the abort protocol for updates generated at any node (except those in the top cluster) that abort. Therefore,

$V_{OW_abort,cpu_i} = pr_quo_i * (n - n_root)(1 - pr_commit)$

$\star$ Computing number of visits to transceiver $k$, connecting nodes $i$ to $j$:

- if $P_i = P_j = 0$ (Node $i$ and $j$ are in the top cluster)

  When $i$ acts as a CO, it sends two messages to $j$ if $j$ belongs to the quorum: vote_req and commit, and it sends one propagation message to $j$ if $j$ is not in the quorum provided that the update commits. Finally, when $j$ acts as a CO, transceiver $k$ processes vote_yes message sent from $i$ to $j$ if $i$ belongs to the quorum. Therefore,

  $$V_{OW_transc,transc_k} = n * pr_co_i * pr_quo_j \mid_{co \neq j} * 2 + pr_co_i(1 - pr_quo_j \mid_{co \neq j}) * (n_root + (n - n_root) * pr_commit) + n * pr_co_j * pr_quo_i \mid_{co \neq i}$$

  It is the same as for Slow_Write with the exception of the second term which applies only if the update commits.

- if $i \in Ch_j$

  $V_{OW_transc,transc_k} = V_{SW_transc,transc_k}$

- if $P_j = i$

  $$V_{OW_transc,transc_k} = 1 + \sum_{s,\forall s \in Ch_j}(1 + descendant_s) + \sum_{w,\forall w \in Nb_j \wedge j \leftrightarrow w \wedge w \not\leftrightarrow i}(1 + descendant_w) + (n_root + (n - n_root) * pr_commit)$$

  Same as $V_{SW_transc,transc_k}$, except for the last term which applies only for updates that commit.

- if $i \in Nb_j \wedge P_i \neq 0 \wedge i \leftrightarrow P_i \wedge j \not\leftrightarrow P_j$

  $$V_{OW_transc,transc_k} = 1 + \sum_{s,\forall s \in Ch_j}(1 + descendant_s) + (n_root + (n - n_root) * pr_commit)$$

  Same as $V_{SW_transc,transc_k}$, except for the last term which applies only for updates that commit.

- if $i \in Nb_j \wedge P_i \neq 0 \wedge j \leftrightarrow P_i \wedge i \not\leftrightarrow P_i$

  $V_{OW_transc,transc_k} = V_{SW_transc,transc_k}$

After having computed $V_{c,k}$ for all job classes at all resources, we can compute the utilisation of the resources:

$U_{c,k} = \lambda_c * V_{c,k} * S_{c,k}$

$U_k = \sum_{i=1}^{C} U_{i,k}$

### 5.1.7 Computation of the response times

In the following, we compute the response times of the different kind of operations, as well as the coverage and reach times for write operations. We address each operation individually.

**Response time of** Fast_Read_0

Since Fast_Read_0 needs just to be processed at the origin node before returning the results to the user, then

$R^i_{F0}$ = Response time of Fast_Read_0 originating at node $i$ $= \frac{S_{F0_proc,cpu_i}}{1-U_{cpu_i}} + \frac{S_{F0_proc,io_i}}{1-U_{io_i}}$

The average response time for Fast_Read_0 is given by: $R_{F0} = \frac{\sum_{i=1}^{n} R^i_{F0}}{n}$

**Response time of** Fast_Read_1

Fast_Read_1 consists of sending the request to the parent, the parent processing the query and then returning back the results. The response time of Fast_read_1 originating at node $i$ can be written as:

$R^i_{F1} = R^i_{F1_proc} + R^i_{F1_comm}$ $\forall i \in NT$, where

$NT$ is the set of nodes that are *not* in the top cluster (It is assumed that Fast_Read_1 of nodes in the top cluster are converted to Fast_Read_0).

$R^i_{F1_proc}$ = the time Fast_Read_1 originating at node $i$ spends in processing

$= \frac{S_{F1_proc,cpu_{P_i}}}{1-U_{cpu_{P_i}}} + \frac{S_{F1_proc,io_{P_i}}}{1-U_{io_{P_i}}}$

$R^i_{F1_comm}$ = the time Fast_Read_1 originating at node $i$ spends in communication

$= comm_{i,P_i} + comm_{P_i,i}$

where $comm_{i,j}$ = the communication time it takes a message to go from node $i$ to $j$

$$= \begin{cases} \frac{S_{F1_comm,cpu_i}}{1-U_{cpu_i}} + \frac{S_{F1_transc,transc_k}}{1-U_{transc_k}} + \frac{S_{F1_comm,cpu_j}}{1-U_{cpu_j}} & \text{if } i \text{ and } j \text{ are directly connected by a } transc_k \\ \frac{S_{F1_comm,cpu_i}}{1-U_{cpu_i}} + lan_del + \frac{S_{F1_comm,cpu_j}}{1-U_{cpu_j}} & \text{if } i \text{ and } j \text{ are directly connected by a lan link} \\ comm_{i,r} + comm_{r,j} & \text{if } i \text{ and } j \text{ are not directly connected and } r \text{ is an intermediate node} \end{cases}$$

The average response time of Fast_read_1 is given by:

$R_{F1} = \sum_{\forall i \in NT} \frac{R^i_{F1}}{|NT|}$

**Response, coverage and reach times of** Fast_Write

**A- The response time**

Since Fast_Write needs just to be processed at the origin node before returning control to the user, then the response time of a Fast_Write originating at node $i$ is given by:

$R^i_{FW} = \frac{S_{FW_proc,cpu_i}}{1-U_{cpu_i}} + \frac{S_{FW_proc,io_i}}{1-U_{io_i}}$

**B- The coverage and reach times**

• $coverage_j$ = average time at which an update originating at any node has been executed at $j$ nodes

• $reach_time_j$ = average time at which an update originating at node $j$ has reached, and has been executed at, all replicas

The two metrics are computed by the algorithm described below, which has the following outlines: assume an update originates at node $i$, then for every node $j$, it computes the time at which the message reaches $j$ ($msg_reach_{i,j}$) and the time at which the update has been executed at node $j$ ($update_reach_{i,j}$).

$msg_reach_{i,j}$ = the time at which an update originated at node $i$ has reached node $j$ (at that point $j$ will start propagating it)

$update_reach_{i,j}$ = the time at which an update originated at node $i$ has been executed at node $j$

The algorithm starts by computing both times for the originating node which are just the processing time of the update. Then, it proceeds by computing $msg_reach_{i,j}$ and $update_reach_{i,j}$ for every subsequent correspondent $j$. The time $msg_reach_{i,j}$ consists of the time the message has reached $i$, plus the time $i$ spends in running the propagation algorithm and the communication time it takes the update to go from $i$ to $j$ (both communication processing time and the time on lines). For $update_reach_{i,j}$, it consists of the time the message reached $j$, plus the time $j$ needs to run the propagation algorithm and to send the message to the subsequent correspondents and finally the processing time of the update at $j$. This works recursively until the time it takes an update originating at node $i$ to reach and be executed at every other node is computed. Then, the row $update_reach_i$ is sorted ascendingly. So, the time at which an update originating at node $i$ reaches all replicas is given by $update_reach_{i,n}$ and $coverage_j$ is computed by accumulating $update_reach_{i,j}$ for all nodes $i$ then getting the average.

```
main() {
    ∀i = 1 to n {
        msg_reach_{i,i} = update_reach_{i,i} = proc(i)
        n_dest = n_neigh_i + n_child_i + n_par_i
        ∀j ∈ Nb_i      reach_neighbour(i, j, n_dest, i)
        ∀s ∈ Ch_i      reach_child(i, s, n_dest, i)
        If P_i ≠ 0     reach_parent(i, P_i, n_dest, i)
        sort_row(update_reach_i)
        ∀j = 1 to n
            coverage_j += update_reach_{i,j}
        reach_time_i = update_reach_{i,n}
    }
    ∀i = 1 to n
        coverage_i = coverage_i / n
}

reach_neighbour(i, j, n_dest, org) {
    /* Compute msg_reach and update_reach for node j when the message is coming from a
neighbour i */
    msg_reach_{org,j} = msg_reach_{org,i} + prop_ovhd(i, n_dest) + comm_{i,j}
    n_dest_j = n_child_j
```

$update_reach_{org,j} = msg_reach_{org,j} + \mathsf{prop_ovhd}(j, n_dest_j) + \frac{direct_child_j * S_{FW_comm,cpu_j}}{1 - U_{cpu_j}} + \mathsf{proc}(j)$

$\forall s \in Ch_j$

reach_child$(j, s, n_dest_j, org)$

}

reach_child$(j, s, n_dest, org)$ {

/* Compute *msg_reach* and *update_reach* for node $s$ when the message is coming from a parent $j$ */

$msg_reach_{org,s} = msg_reach_{org,j} + \mathsf{prop_ovhd}(j, n_dest) + comm_{j,s}$

$n_dest_s = n_child_s$

$update_reach_{org,s} = msg_reach_{org,s} + \mathsf{prop_ovhd}(s, n_dest_s) + \frac{direct_child_s * S_{FW_comm,cpu_s}}{1 - U_{cpu_s}} + \mathsf{proc}(s)$

$\forall s1 \in Ch_s$

reach_child$(s, s1, n_dest_s, org)$

}

reach_parent$(i, p, n_dest, org)$ {

/* Compute *msg_reach* and *update_reach* for node $p$ when the message is coming from a child $i$ */

$msg_reach_{org,p} = msg_reach_{org,i} + \mathsf{prop_ovhd}(i, n_dest) + comm_{i,p}$

$n_dest_p = n_neigh_p + n_par_p + \mid SC_{p,i} \mid$

$n_direct_dest_p = direct_neigh_p + direct_par_p + \mid DC_{p,i} \mid$

$update_reach_{org,p} = msg_reach_{org,p} + \mathsf{prop_ovhd}(p, n_dest_p) + \frac{n_direct_dest_p * S_{FW_comm,cpu_p}}{1 - U_{cpu_p}} + \mathsf{proc}(p)$

$\forall j \in Nb_p$

reach_neighbour$(p, j, n_dest_p, org)$

$\forall s \in Ch_p \wedge C_s \neq C_i$

reach_child$(p, s, n_dest_p, org)$

If $P_p \neq 0$ reach_parent$(p, P_p, n_dest_p, org)$

}

proc$(j)$ {

/* Compute the time it takes an update to be processed at node $j$ */

return$(\frac{S_{FW_proc,cpu_j}}{1 - U_{cpu_j}} + \frac{S_{FW_proc,io_j}}{1 - U_{io_j}})$

}

prop_ovhd$(i, n_dest)$ {

/* Compute the time it takes node $i$ to execute the overhead of the propagation algorithm sending the message to $n_dest$ logical destinations */

return$(\frac{S_{FW_const_propg_ovhd,cpu_i} + S_{FW_var_propg_ovhd,cpu_i} * n_dest}{1 - U_{cpu_i}})$

}

**Response, coverage and reach time of** Slow_Write

**A- The response time**

An approximate solution has been used to estimate the average response time of synchronous operations. The solution follows the approach presented in [CDY90]. It partitions the analysis into two components: resource contention and data contention. The *Mean Value Analysis* (MVA) approach is used to estimate the hardware resource contention effects as was described previously. The estimation of the lock contention wait time is presented below. In our model, data contention between operations is mapped into lock waits rather than aborts. Therefore, abort occurs only in case of deadlocks. However, the overhead of deadlock and abort is not considered in the analysis due to its complexity.

A Slow_Write consists of sending the request to CO, CO assembles a quorum from its neighbours, then CO sends commit to the origin node and finally the origin node processes the update before returning control to the user. So, the response time of a Slow_Write originating at node $i$ can be written as:

$R^i_{SW} = R^i_{SW_comm} + R^i_{SW_commit} + R^i_{SW_proc}$, where

$R^i_{SW_comm}$ = communication time for $i$ to send the request to and receive commit from $CO_i$

$$= \begin{cases} comm_{i,CO_i} + comm_{CO_i,i} & \text{if } P_i \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$R^i_{SW_proc}$ = the processing time of the update at node $i$

$$= \frac{S_{SW_proc,cpu_i}}{1-U_{cpu_i}} + \frac{S_{SW_proc,io_i}}{1-U_{io_i}}$$

$R^i_{SW_commit}$ = the time it takes $CO_i$ to commit the update

$$= R^i_{SW_commit_no_cont} + R^i_{SW_cont}$$

where $R^i_{SW_commit_no_cont}$ is the time it takes the update to commit provided there is no contention, i.e. the update finds the locks free. It includes the CPU time at $CO_i$ to perform the 2PC algorithm (phase 1 and 2), the time it takes $CO_i$ to send vote_req to participants and receive vote_yes (which consists of the communication delays to send vote_req and receive vote_yes and the CPU time at participants to acquire locks), and finally the CPU time at $CO_i$ to send commit messages to all participants before sending commit to origin. It is given by:

$$R^i_{SW_commit_no_cont} = R^{CO_i}_{ovhd_ph1} + \frac{\sum_{\forall Q_j \in \Pi_{CO_i}} \mathsf{Max}_{\forall q \in Q_j}\{comm_{CO_i,q} + R^q_{cpu_lock} + comm_{q,CO_i}\}}{|\Pi_{CO_i}|} + R^{CO_i}_{ovhd_ph2} + R^{CO_i}_{cpu_comm_q}, \quad \text{where}$$

$R^{CO_i}_{ovhd_ph1} = R^{CO_i}_{ovhd_ph2}$ = CPU time to execute one phase of 2PC algorithm at $CO_i$

$$= \frac{S_{SW_const_2pc_ovhd,cpu_{CO_i}} + q_size * S_{SW_var_2pc_ovhd,cpu_{CO_i}}}{1-U_{cpu_{CO_i}}}$$

$R^q_{cpu_lock}$ = CPU time to acquire or release a lock at a participant $q$

$$= \frac{S_{SW_lock,cpu_q}}{1-U_{cpu_q}}$$

$R^{CO_i}_{cpu_comm_q}$ = the time $CO_i$ spends on CPU sending commit messages to participants

$$= \frac{S_{SW_comm,cpu_{CO_i}} * (q_size - 1)}{1-U_{cpu_{CO_i}}}$$

$\Pi_{CO_i}$ is the set containing all possible sets of participants for node $CO_i$

$R^i_{SW_cont}$ = the time spent in contention wait by an operation originating at node $i$ for a lock that is held by another operation

The approach used to evaluate $R_{SW_cont}$ is the same as in [CDY90], which is based on the intuition that if a transaction $T_1$ conflicts with another transaction $T_2$, then $T_1$ has to wait for $T_2$ to release its locks. Consider:

$R^j_{hold_lock_\alpha}$ = total amount of time during which locks are held by an operation of type $\alpha$ generated by node $j$.

$\alpha_pr_conflict_j$ = probability that an operation will conflict with an operation of type $\alpha$ generated by node $j$

Since Slow_Write conflicts with all Slow_Write, Slow_Read and Opt_Write operations, then

$$R^i_{SW_cont} = \sum_{j=1}^{n} SW_pr_conflict_j * R^j_{hold_lock_SW} * \gamma_{SW} + \sum_{j=1}^{n} SR_pr_conflict_j * R^j_{hold_lock_SR} * \gamma_{SR} + \sum_{j=1}^{n} OW_pr_conflict_j * R^j_{hold_lock_OW} * \gamma_{OW}$$

where $\gamma_\alpha$ is a contention factor

The conflict probabilities are given by:

$$SW_pr_conflict_j = \lambda_j * (1 - freq_r) * slow_w * R^j_{hold_lock_SW}$$
$$SR_pr_conflict_j = \lambda_j * freq_r * slow_r * R^j_{hold_lock_SR}$$
$$OW_pr_conflict_j = \lambda_j * (1 - freq_r) * opt_w * R^j_{hold_lock_OW}$$

$R^j_{hold_lock_SW}$ = Total amount of time during which locks are held by a Slow_Write generated by node $j$

$$= \frac{\sum_{\forall Q_k \in \Pi_{CO_j}} \mathsf{Max}_{\forall q \in Q_k}\{comm_{CO_j,q} + R^q_{cpu_lock} + comm_{q,CO_j}\}}{|\Pi_{CO_j}|} + R^{CO_j}_{ovhd_ph2} + \mathsf{Max}\{R^{CO_j}_{cpu_comm_q} + R^{CO_j}_{cpu_comm_org} + R^{CO_j}_{SW_proc} + R^{CO_j}_{cpu_lock}, \frac{\sum_{\forall Q_k \in \Pi_{CO_j}} \mathsf{Max}_{\forall q \in Q_k \wedge q \neq CO_j}\{comm_{CO_j,q} + R^q_{SW_proc} + R^q_{cpu_lock}\}}{|\Pi_{CO_j}|}\}$$

The first term accounts for the time it takes $CO_j$ to send vote_req to participants and to receive vote_yes while holding locks. The second term accounts for locks being held while $CO_j$ performs the second phase of 2PC. Finally, the locks are released after: either the time for $CO_j$ to send commit to participants and origin nodes, process the update and release locks, or the time for a participant to receive a commit from $CO_j$, process the update and release locks whichever is greater.

$R^j_{hold_lock_SR}$ = Total amount of time during which locks are held by a Slow_Read generated by node $j$.

$$= \frac{\sum_{\forall Q_k \in \Pi_{CO_j}} \mathsf{Max}_{\forall q \in Q_k}\{comm_{CO_j,q} + R^q_{cpu_lock} + comm_{q,CO_j}\}}{|\Pi_{CO_j}|} + R^{CO_j}_{ovhd_ph2} + \frac{\mathsf{Max}\{R^{CO_j}_{cpu_comm_q} + R^{CO_j}_{SR_proc} + R^{CO_j}_{cpu_lock}, \frac{\sum_{\forall Q_k \in \Pi_{CO_j}} \mathsf{Max}_{\forall q \in Q_k \wedge q \neq CO_j}\{comm_{CO_j,q} + R^q_{cpu_lock}\}}{|\Pi_{CO_j}|}\}}{q_size}$$

$$+\frac{(q_size-1)*\mathsf{Max}\{R^{CO_j}_{cpu_comm_q}+R^{CO_j}_{cpu_lock},\frac{\sum_{\forall q\in Nb_{CO_j}} comm_{CO_j,q}+R^q_{SR_proc}+R^q_{cpu_lock}}{n_neigh_{CO_j}}\}}{q_size}$$

The first two terms are the same as for Slow_Write. The third term considers the case when $CO_j$ is the node selected to perform the read. In this case, locks are released either after $CO_j$ sends commit messages to the participants, processes the read and releases locks, or after the time it takes a participant to receive commit from $CO_j$ and release locks whichever is greater. The fourth term considers the case when a node from the quorum is selected to perform the read, then locks are released either after $CO_j$ sends commit messages to the participants and release locks or after the selected node receives the order from $CO_j$ to perform the read, executes it then releases locks, whichever is greater.

$R^j_{hold_lock_OW}$ = Total amount of time during which locks are held by an Opt_Write generated by node $j$

$$= \frac{\sum_{\forall Q_k\in\Pi_{CO_j}} \mathsf{Max}_{\forall q\in Q_k}\{comm_{CO_j,q}+R^q_{cpu_lock}+comm_{q,CO_j}\}}{|\Pi_{CO_j}|}+R^{CO_j}_{ovhd_ph2}$$
$$+\mathsf{Max}\{R^{CO_j}_{cpu_comm_q}+R^{CO_j}_{cpu_comm_org}+R^{CO_j}_{OW_proc}+R^{CO_j}_{cpu_lock},$$
$$\frac{\sum_{\forall Q_k\in\Pi_{CO_j}} \mathsf{Max}_{\forall q\in Q_k\wedge q\neq CO_j}\{comm_{CO_j,q}+R^q_{OW_proc}+R^q_{cpu_lock}\}}{|\Pi_{CO_j}|}\}*pr_commit$$
$$+\mathsf{Max}\{R^{CO_j}_{cpu_comm_q}+R^{CO_j}_{cpu_comm_org}+R^{CO_j}_{cpu_abort}+R^{CO_j}_{cpu_lock},$$
$$\frac{\sum_{\forall Q_k\in\Pi_{CO_j}} \mathsf{Max}_{\forall q\in Q_k\wedge q\neq CO_j}\{comm_{CO_j,q}+R^q_{cpu_abort}+R^q_{cpu_lock}\}}{|\Pi_{CO_j}|}\}*(1-pr_commit)$$

where $R_{cpu_abort}=\frac{S_{OW_abort,cpu_j}}{1-U_{cpu_j}}$

The first two terms are the same as Slow_Write. The next two terms accounts for the time at which locks are released once $CO_j$ has executed Phase 2 of 2PC. If the operation commits, then it is the same as for Slow_Write which occurs with probability $pr_commit$. If the operation aborts, then the locks are released after either the time it takes $CO_j$ to send abort to the participants and origin nodes, execute the abort protocol and release locks, or the time it takes a participant to receive abort from CO, execute the abort protocol and release locks whichever is greater. This occurs with probability $(1-pr_commit)$.

Since locks are obtained at different times by each participant during the first phase and they are released at different times depending on when they receive the commit and since the arriving update can conflict with another request while the later is in the first phase, second phase or just before releasing locks, then, a request might not need to wait for the whole $R^j_{hold_lock_\alpha}$. Therefore, $R_{SW_cont}$ is multiplied by a factor $\gamma$. In [CDY90], $\gamma$ was taken to be 0.5 based on the heuristic that: locks are obtained in the beginning of Phase 1, then the average wait time for such a contention is estimated to be $R^j_{hold_lock_\alpha}/2$. However, taking $\gamma$ as absolute value 0.5 does not take into consideration the degree of contention: if there is a little contention, then it is plausible to assume that $\gamma=0.5$. But if the degree of the contention is high and an incoming request has to wait for few requests to complete, then it should wait for the whole $R^j_{hold_lock_\alpha}$.

Therefore, the value of $\gamma$ is taken proportional to the degree of contention

$$\gamma_\alpha=\begin{cases}0.5 & \text{if } \beta_\alpha<0.5\\ \beta_\alpha & \text{otherwise}\end{cases}$$

where,

$\beta_{SW} = \sum_{j=1}^{n} SW_pr_conflict_j + SR_pr_conflict_j + OW_pr_conflict_j$

$\beta_{SW}$ represents the degree of contention that a Slow_Write experiences since it conflicts with Slow_Write, Slow_Read and Opt_Write.

$\beta_{SR} = \sum_{j=1}^{n} SW_pr_conflict_j + OW_pr_conflict_j$

$\beta_{SR}$ represents the degree of contention that a Slow_Read experiences since it conflicts with Slow_Write and Opt_Write.

$\beta_{OW} = \beta_{SW}$

Therefore, if there is not much contention, ($\beta < 0.5$), a job waits on the average for $R_{hold_lock}/2$ time units. If there are lots of contention, then $\beta$ increases and the contention time increases accordingly.

Finally, the average response time of Slow_Write is given by:

$$R_{SW} = \sum_{i=1}^{n} \frac{R^i_{SW}}{n}$$

**B- The coverage and reach times**

The computation of the reach and coverage times is similar to that of Fast_Write with a slight modification to the equations of $msg_reach_{i,j}$ and $update_reach_{i,j}$ for nodes that participate in the quorum. Those nodes start propagating the message only after executing the update and releasing locks.

$CO_i$ starts propagating a Slow_Write generated at node $i$ after the request reaches $CO_i$, $CO_i$ commits, executes the update and releases locks. Therefore,

$$\begin{aligned} msg_reach_{i,CO_i} &= update_reach_{i,CO_i} \\ &= comm_{i,CO_i} + R^i_{SW_commit} + R^{CO_i}_{SW_proc} + R^{CO_i}_{cpu_lock} \end{aligned}$$

Similarly, for a node $q$ that participates in the quorum:

$$\begin{aligned} msg_reach_{i,q} &= update_reach_{i,q} \\ &= comm_{i,CO_i} + R^i_{SW_commit} + comm_{CO_i,q} + R^q_{SW_proc} + R^q_{cpu_lock} \end{aligned}$$

The algorithm is similar to that of Fast_Write.

**Response time of Slow_Read**

Slow_Read consists of sending the request to CO, CO assembles a quorum from its neighbours, then either the CO executes the read or it selects a node from the quorum to execute it. In the later case, the node sends back the results to CO which forwards it to the origin node. The response time of a Slow_Read originated at node $i$ can be written as:

$R^i_{SR} = R^i_{SR_comm} + R^i_{SR_commit}$, where

$R^i_{SR_comm}$ = communication time for $i$ to send the request to and receive results from CO

$$= \begin{cases} comm_{i,CO_i} + comm_{CO_i,i} & \text{if } P_i \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$R^i_{SR_commit}$ = the time it takes $CO_i$ to commit the read

$$= R^i_{SR_commit_no_cont} + R^i_{SR_cont}$$

where $R^i_{SR_commit_no_cont}$ is the time it takes the read to commit provided there is no contention, i.e. all locks are free. It includes the CPU time at $CO_i$ to perform the 2PC algorithm (phase 1 and 2), and the time it takes CO to send vote_req to participants and receive vote_yes. In the case $CO_i$ executes the read, which is with probability $1/q_size$, the operation commits

after $CO_i$ sends commit messages to all participants, executes the read and releases locks. In case a participant is selected to perform the read, then the operation commits after the selected node receives the order from $CO_i$ to execute the read, executes it, releases locks and sends results to $CO_i$. Assuming that a node from the quorum will execute the read with equal probability, and since any neighbour of $CO_i$ can be selected to participate in the quorum, then an average over the neighbours is computed for the last term.

$$R^i_{SR_commit_no_cont} = R^{CO_i}_{ovhd_ph1} + \frac{\sum_{\forall Q_j \in \Pi_{CO_i}} \mathsf{Max}_{\forall q \in Q_j}\{comm_{CO_i,q} + R^q_{cpu_lock} + comm_{q,CO_i}\}}{|\Pi_{CO_i}|} + R^{CO_i}_{ovhd_ph2}$$
$$+ \frac{R^{CO_i}_{cpu_comm_q} + R^{CO_i}_{SR_proc} + R^{CO_i}_{cpu_lock}}{q_size}$$
$$+ \frac{q_size-1}{q_size} * \frac{\sum_{\forall j \in Nb_{CO_i}} comm_{CO_i,j} + R^j_{SR_proc} + R^j_{cpu_lock} + comm_{j,CO_i}}{n_neigh_{CO_i}}$$

Since Slow_Read conflicts with Slow_Write and Opt_Write but not with Slow_Read, then

$R^i_{SR_cont}$ = the time spent in contention wait by a Slow_Read originating at node $i$ for a lock that is held by an incompatible operation

$$R^i_{SR_cont} = \sum_{j=1}^{n} SW_pr_conflict_j * R^j_{hold_lock_SW} * \gamma_{SW}$$
$$+ \sum_{j=1}^{n} OW_pr_conflict_j * R^j_{hold_lock_OW} * \gamma_{OW}$$

**Response, coverage and reach time of** Opt_Write

**A- The response time**

A node $i$ originating an Opt_Write executes the update and sends it to $CO_i$. Nodes on the path to $CO_i$ are considered *optimistic node* and execute the update as well. After $CO_i$ assembles a quorum from its neighbours, it sends a commit or abort message to the origin node. The response time of Opt_Write is basically the time it takes the origin node to process the update:

$$R^i_{OW} = R^i_{OW_proc} = \frac{S_{OW_proc,cpu_i}}{1-U_{cpu_i}} + \frac{S_{OW_proc,io_i}}{1-U_{io_i}}$$

$R^i_{OW_commit}$ = the time it takes $CO_i$ to commit the update (is the same as for Slow_Write)

$$= R^i_{SW_commit}$$

The time at which an origin node knows whether the update has committed or aborted = $R^i_{OW_commit} + R^i_{OW_comm}$, where

$R^i_{OW_comm}$ = communication time for $i$ to send request to and receive commit or abort from $CO_i$

$$= R^i_{SW_comm}$$

**B- The coverage and reach times**

For committed requests, the computation of the reach and coverage times is similar to that of Fast_Write with some modifications to the equations of $msg_reach_{i,j}$ and $update_reach_{i,j}$ for nodes that participate in the quorum as well as optimistic nodes.

For nodes that participate in the quorum, the modifications are the same as for Slow_Write. Consider a node $i$ originating an Opt_write, then, for the $CO_i$:

$$msg_reach_{i,CO_i} = update_reach_{i,CO_i}$$
$$= comm_{i,CO_i} + R^i_{OW_commit} + R^{CO_i}_{OW_proc} + R^{CO_i}_{cpu_lock}$$

For a node $q$ that participates in the quorum:

$msg_reach_{i,q} = update_reach_{i,q}$

$= comm_{i,CO_i} + R^i_{OW_commit} + comm_{CO_i,q} + R^q_{OW_proc} + R^q_{cpu_lock}$

For the origin node $i$, since it executes the update upon originating it, then

$update_reach_{i,i} = R^i_{OW_proc}$

Since $i$ starts propagating the message after it receives the commit message from $CO_i$, then

$msg_reach_{i,i} = msg_reach_{i,CO_i} + comm_{CO_i,i}$

An optimistic node $j$ executes the update after $i$ executes it and sends it to $j$. Also, $j$ starts propagating the update after it receives the commit from $CO_i$. Therefore,

$update_reach_{i,j} = R^i_{OW_proc} + comm_{i,j} + R^j_{OW_proc}$

$msg_reach_{i,j} = msg_reach_{i,CO_i} + comm_{CO_i,j}$

### 5.1.8 Discussion

In the simulation, the queueing discipline used for CPU, disks and transceivers is FCFS with no preemption. While deriving the analytical output equations for $U_{c,k}$ and $R_{c,k}$ [LZGS84], it was necessary to impose the restriction that $S_{c,k}$ is the same for all classes. Since in our model, $S_{c,k}$ is different for different classes, we have assumed that $S^*_{c,k} = 1$ (or shortest job length) and that $V^*_{c,k} = S_{c,k} * V_{c,k}$ for the solution to be applicable. Since the output equations contain only the term $D_{c,k}$ then the final equations remain unchanged.

However, this means that the analytical solution treats the scheduling discipline more or less like *Round Robin* with quantum =1 (or shortest job length), or *processor sharing* rather than FCFS in the presence of unequal jobs. For instance, short jobs, such as processing communication messages, would have to wait for $S^*_{c,k}$ ( which would be equal to 1 or shortest job length) in the presence of longer jobs, such as *cpu_req*, rather than for a whole read or write request. This gives an advantage to short jobs over long jobs. Therefore, the estimated time short jobs spend in the system is less than what is obtained by simulation.

So, the time spent by communication and overhead jobs at the CPU are expected to be less than simulation results while the processing time of the read and write requests is expected to be higher analytically, as they are treated as long jobs. As for the communication time spent on transceivers or the processing time of the IO requests, they are equal for both models since jobs queueing at these resources are of the same length.

As was mentioned before, the effect of deadlock is modelled in simulation but not analytically due to its complexity. This makes a deviation in the results which tends to let the analytical estimation times less than what is obtained simulation, especially when the contention is high as the possibility of deadlocks increases.

For simplicity of analysis, it is assumed that nodes in the top cluster are always fully connected. This is to ignore the effect of intermediate neighbours as it is hard to determine analytically which nodes belongs to the quorum and which do not since they are chosen randomly. This assumption is not present in the simulation model. Thus some deviation would result in certain cases (e.g. Net 6) making the analytical results lower.

For clusters other than the top clusters, nodes do not have to be fully connected. However, in the analysis it is assumed that there is one level of indirection between a child and its parent or its neighbours These properties apply to all the physical structures evaluated in this study.

## 5.2 Results of the verification

In this section, we have selected some experiments and we compare the results obtained by both the simulation model and the analytical model.

### 5.2.1 Varying read mix when writes are Slow_Write

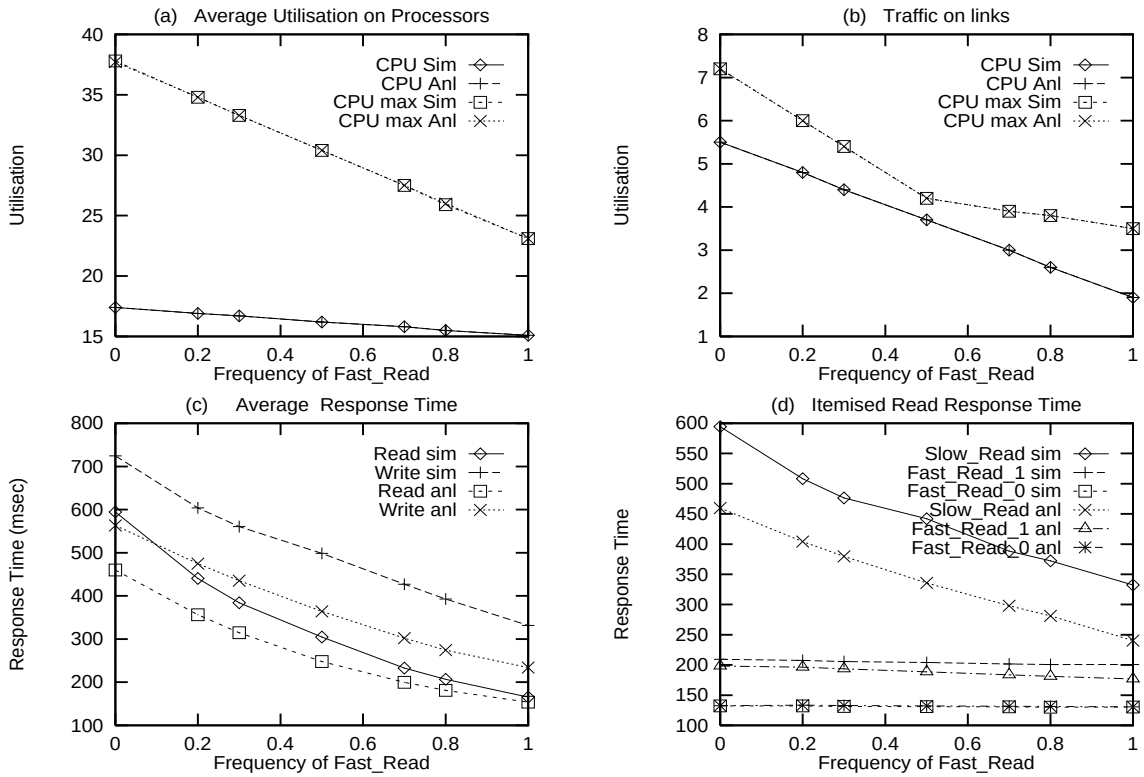


Figure 35: Varying read mix, writes are Slow_Writes

As shown in Figure 35 (a) and (b), the utilisation of the resources obtained by the simulation and the analytical model is identical.

The analytical response time of writes (Slow_Write), as shown in Figure 35 (c), follows the same curve as the simulation but is constantly lower. This is because Slow_Write involves many short jobs, such as the overhead of 2PC, acquiring and releasing locks, communication processing and propagation protocol. These short jobs are given higher priority over long jobs (processing requests) in the analytical model due to the approximation used, which is the not the case in simulation. Further, the effect of deadlock and abort are not modelled analytically while they are considered in simulation. Therefore, the write response time analytically is lower than that obtained from simulation.

Similarly, the average read response time is lower in the analytical results. However, the gap decreases as the frequency of Fast_Read increases as shown. This is more illustrated in

Figure 35 (d) which plots the response time of individual read operations. We observe that the response time of Slow_Read obtained analytically is constantly lower than that of simulation. The reasons are the same as for Slow_Write. The response of Fast_Read_0 is nearly the same for both models. While the response of Fast_Read_1 is slightly lower analytically. This is because Fast_Read_1 involves communication processing overhead which is given higher priority in the analytical model and is estimated to stay less in the system. Therefore, as $fast_r$ increases, the response of Fast_Read dominates and the gap between analytical and simulation results becomes smaller as shown in Figure 35 (c).

### 5.2.2 Varying the communication processing overhead, writes are synchronous

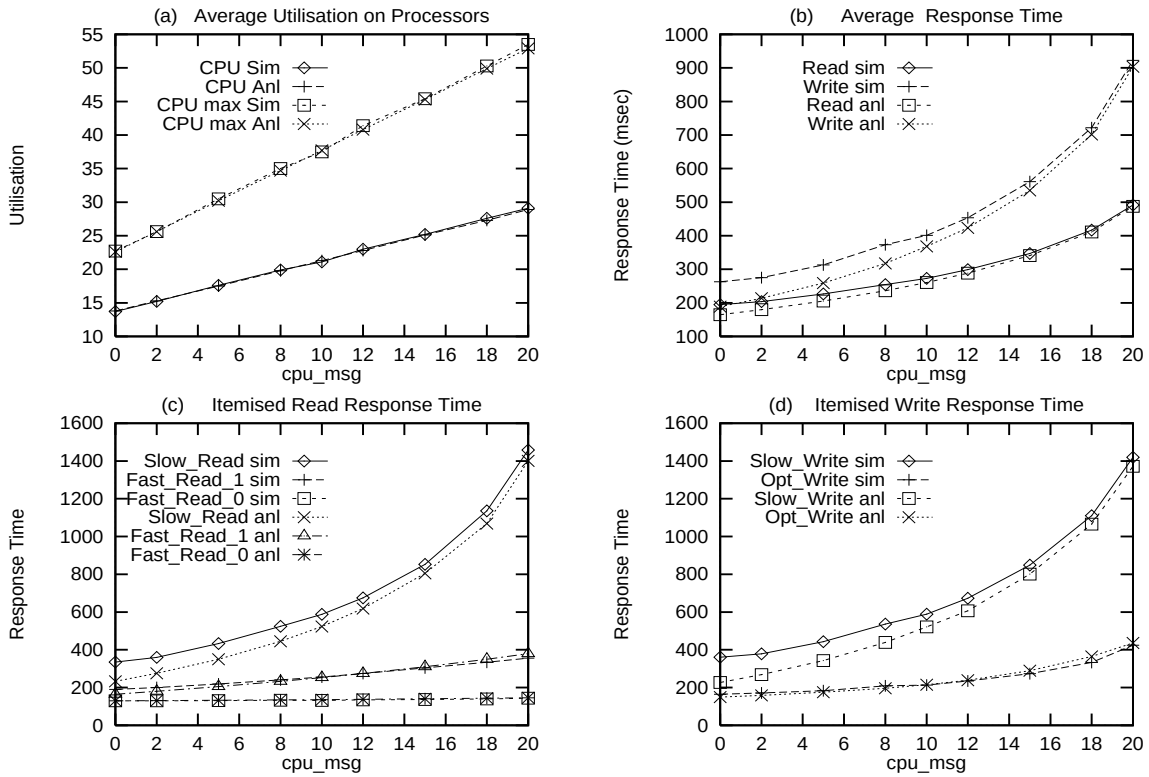


Figure 36: Varying communication processing overhead

Utilisation of the processors is plotted in Figure 36 (a) and as shown they are identical in both analytical and simulation. The same applies for the utilisation of the IO and communication devices.

The response time of reads and writes (Figure 36 (b)) obtained by the analytical model are lower than those of simulation for low values of *cpu_msg*. But the gap decreases as *cpu_msg* increases. This is more illustrated in Figures 36 (c) and (d) which plot the response time of individual read and writes operations. For read operations, we observe that the response time of Fast_Read_0 is the same for both. Response of Fast_Read_1 is slightly lower analytically for

small values of *cpu_msg*. This is because communication jobs are treated with high priority analytically, hence, the time the operation spends in the system is less than what is obtained by simulation. But as *cpu_msg* increases, shorter jobs (such as 2PC and propagation overhead) are given higher priority than communication jobs and therefore the response time of Fast_Read_1 increases and bypasses that of simulation very slightly. Similarly, the response time of Slow_Read analytically is lower than simulation for low values of *cpu_msg* then the gap decreases gradually as communication jobs are treated as long jobs. However, the gap does not disappear as Slow_Read involves many other short jobs and also the effect of deadlock and abort is not considered. The same comments apply for Slow_Write as shown in Figure 36 (d). Since nodes in the top cluster their Opt_Write are converted to Slow_Write, so the response time of Opt_Write is a mix between the response time of processing the requests (for nodes not in the top cluster) and the response time of Slow_Write. It is slightly lower than simulation for low *cpu_msg*, when the decrease in the response of Slow_Write dominates. Then, it increases and becomes slightly higher, when the increase in the processing time of requests dominates.

### 5.2.3 Varying the network topology (Net 2)

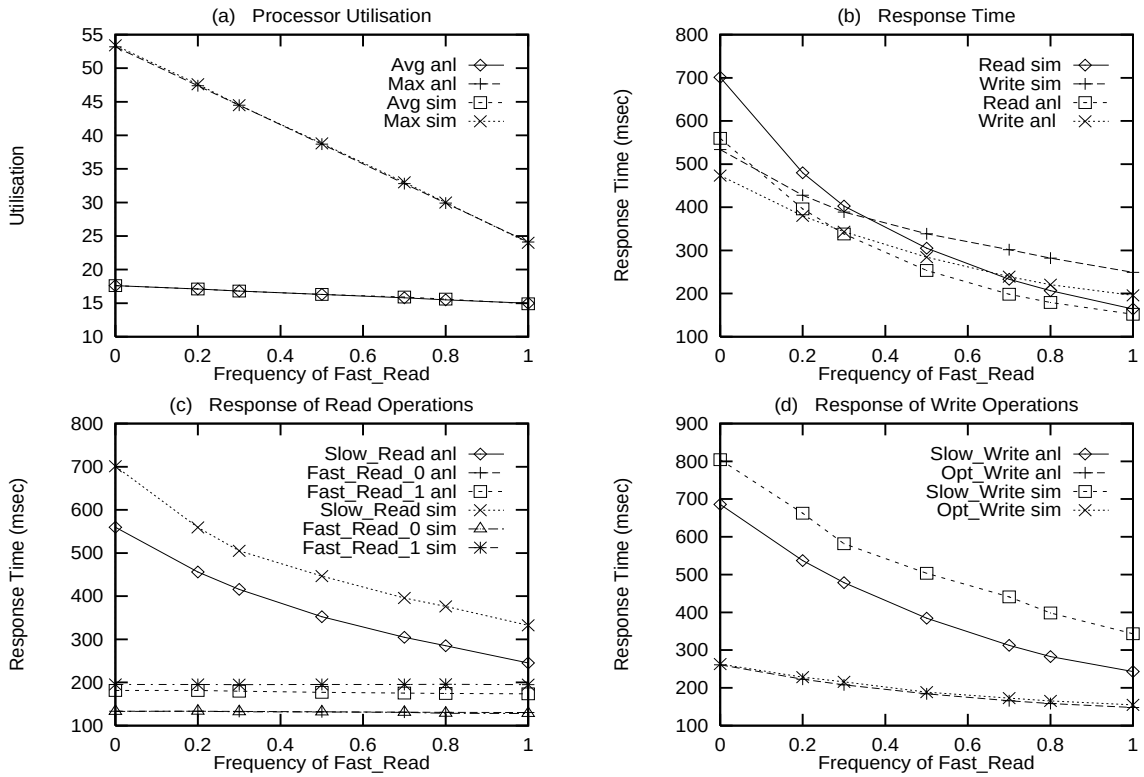


Figure 37: Varying the read mix, network topology (Net 2)

In this section, we verify the results obtained from the simulator under different network topologies. The network Net 2 is chosen as an example. The results of varying the read mix when

updates are synchronous are plotted in Figure 37.

The same comments on Net 1 apply to Net 2.

The utilisation of the resources is the same in both analytical and simulation models. The utilisation of the processors is plotted in Figure 37 (a).

As shown in Figure 37 (c), the response time of Fast_Read_0 is identical in both models. That of Fast_Read_1 is slightly lower analytically due to communication jobs. The response of Slow_Read is lower than simulation for the same reasons mentioned before: the presence of short jobs which are estimated analytically to stay in the system less than what is obtained from the simulation, and the fact that deadlocks and aborts are not modelled). Consequently, as shown in Figure 37 (b), the average read response time analytically is lower than that of simulation but the gap decreases as the frequency of Fast_Read increases.

Similar to Slow_Read, the response of Slow_Write is lower analytically while the response of Opt_Write is nearly identical as shown in Figure 37 (d). Consequently, the average response time of write is a mix of these two responses as shown in Figure 37 (c).

### 5.2.4 Varying the mix between reads and writes, writes are Fast_Writes

As shown in Figures 38 (a) and (b), the utilisation of the resources is identical for both simulation and analytical models.

Due to the approximation used in the analytical model, the estimated time spent by the operations in processing is higher than what is obtained by simulation, while the time spent in communication processing and propagation overhead is lower analytically. Therefore, we observe that the response time of Fast_Write and Fast_Read_0 operations (Figures 38 (c) and (d) respectively) is higher analytically since they consist mainly of processing the operation. The gap gets bigger as the frequency of writes increases since the resources become more loaded.

The response time of Fast_Read_1, Figure 38 (d), analytically is slightly lower than simulation when resources are not loaded. This is because the reduction in the estimated communication time is larger than the increase in the estimated processing time. As the resources get more contended, the processing time increases and causes the response time estimated analytically to bypass that of simulation.

The reach time, as shown in Figure 38 (e), analytically follows the same curve as that of simulation but is constantly lower. This is because it involves a fair amount of communication processing and processing propagation overhead which are estimated lower analytically. The coverage time, Figure 38 (f), is higher analytically for the first few nodes, where the processing time is dominating. Then, the analytical curve decreases and becomes lower than that of simulation when the reduction in the estimated communication time and propagation time overhead dominates. The differences are bigger for higher frequency of writes when resources are more loaded.

### 5.2.5 Varying the communication processing overhead, writes are Fast_Write

Varying the communication processing overhead when updates are Fast_Write, we observe that the utilisation of the resources is the same in both models (Figure 39 (a)).

The response time of Fast_Write and Fast_Read_0, as shown in Figures 39 (b) and (c) are slightly higher analytically but the gap decreases as communication jobs are longer. The response time of Fast_Read_1 (Figure 39(c)) is lower analytically for low values of *cpu_msg*. This is because

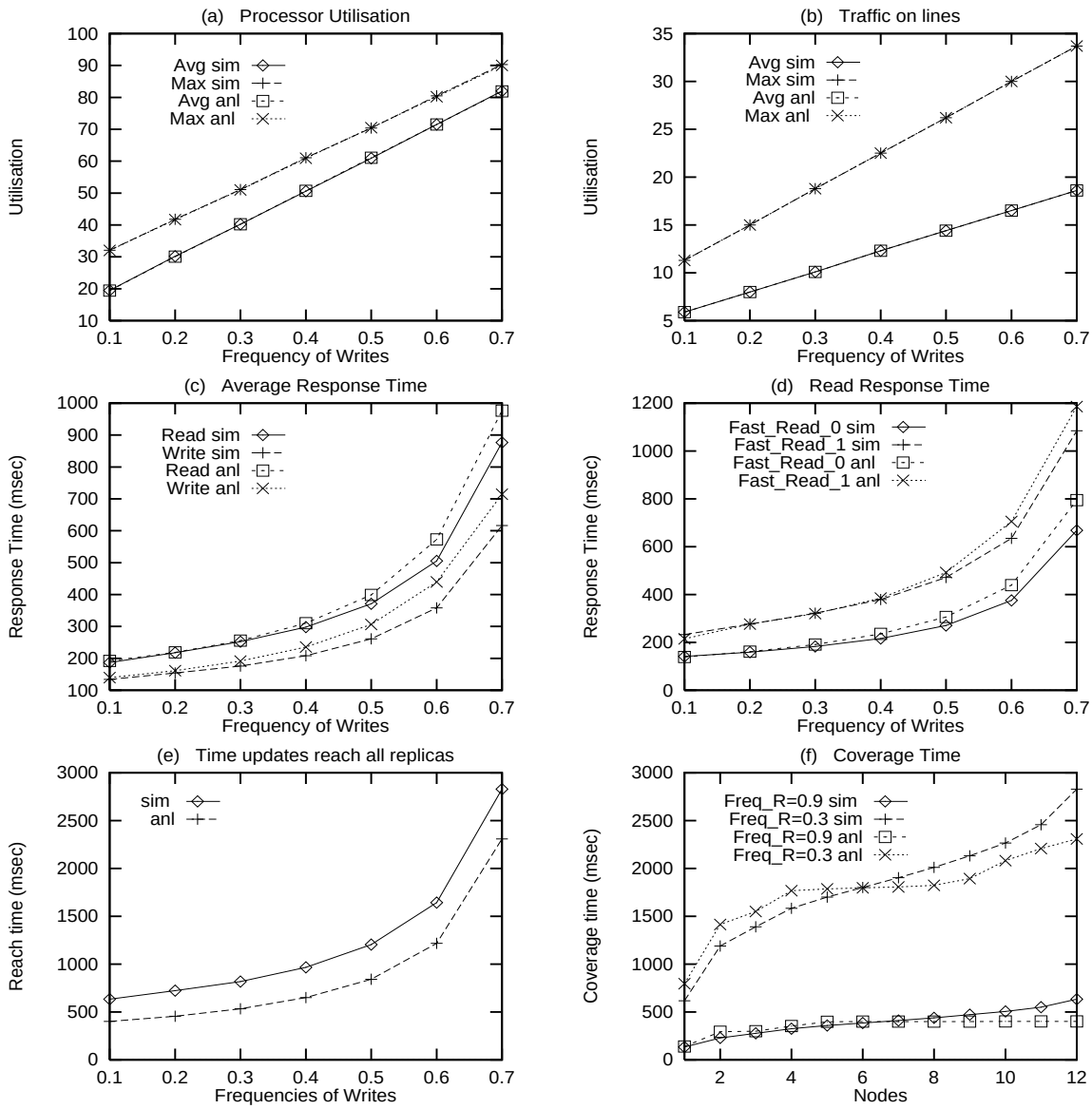


Figure 38: Varying the read mix, writes are Fast_Write

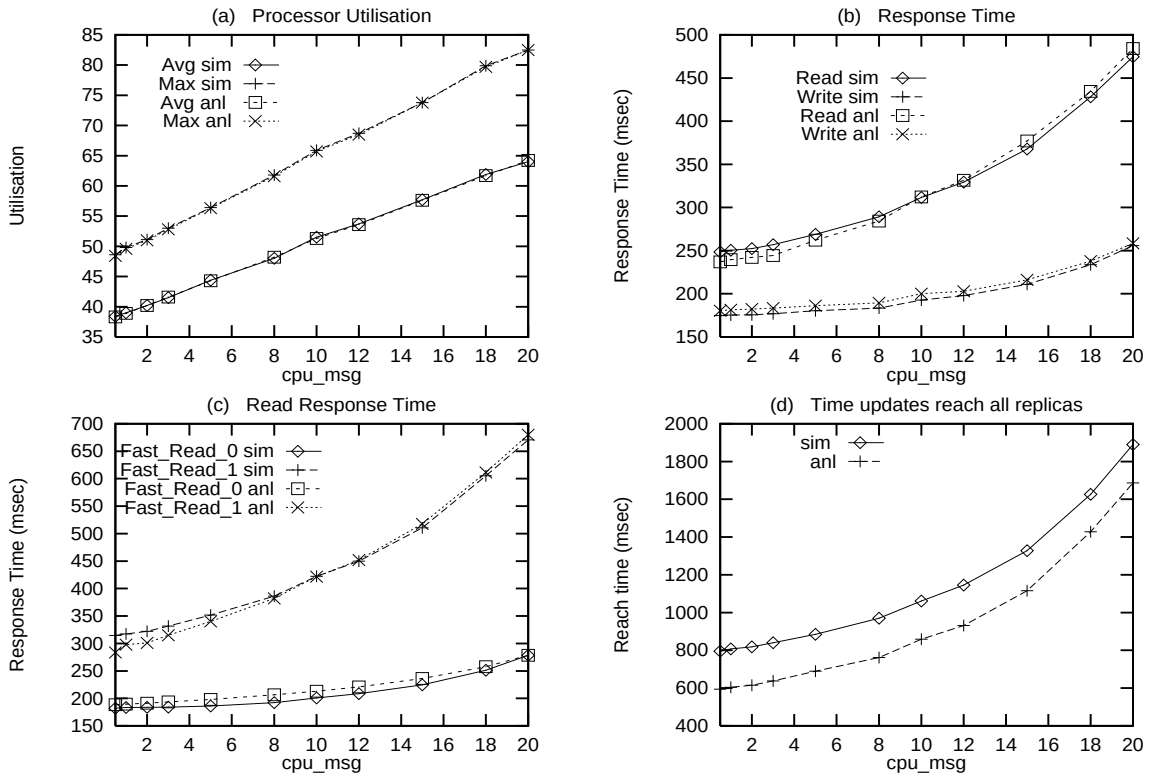


Figure 39: Varying communication processing overhead, writes are Fast_Write

the reduction in the estimated communication time is larger than the increase in the estimated processing time. As *cpu_msg* increases, shorter jobs (such as propagation overhead) are given higher priority than communication jobs and therefore the response time of Fast_Read_1 increases and bypasses that of simulation very slightly. The analytical reach time follows the same curve as in simulation but is constantly lower due to the reduction in the estimated communication time. However, the gap decreases as the communication jobs get longer.

### 5.2.6 Varying the read mix, comparing Net 7 and Net 9, writes are synchronous

The same comments applied to Net 1 apply to Net 7 and Net 9:

Utilisation of the resources is the same in both the simulation and analytical model.

For both Net 7 and Net 9, the response time of Opt_Write and Fast_Read_0 is identical in the simulation and analytically as shown in Figure 40. The response time of Fast_Read_1 is slightly lower analytically. The response time of Slow_Write and Slow_Read are constantly lower analytically by around 100 msec, for the same reasons mentioned before.

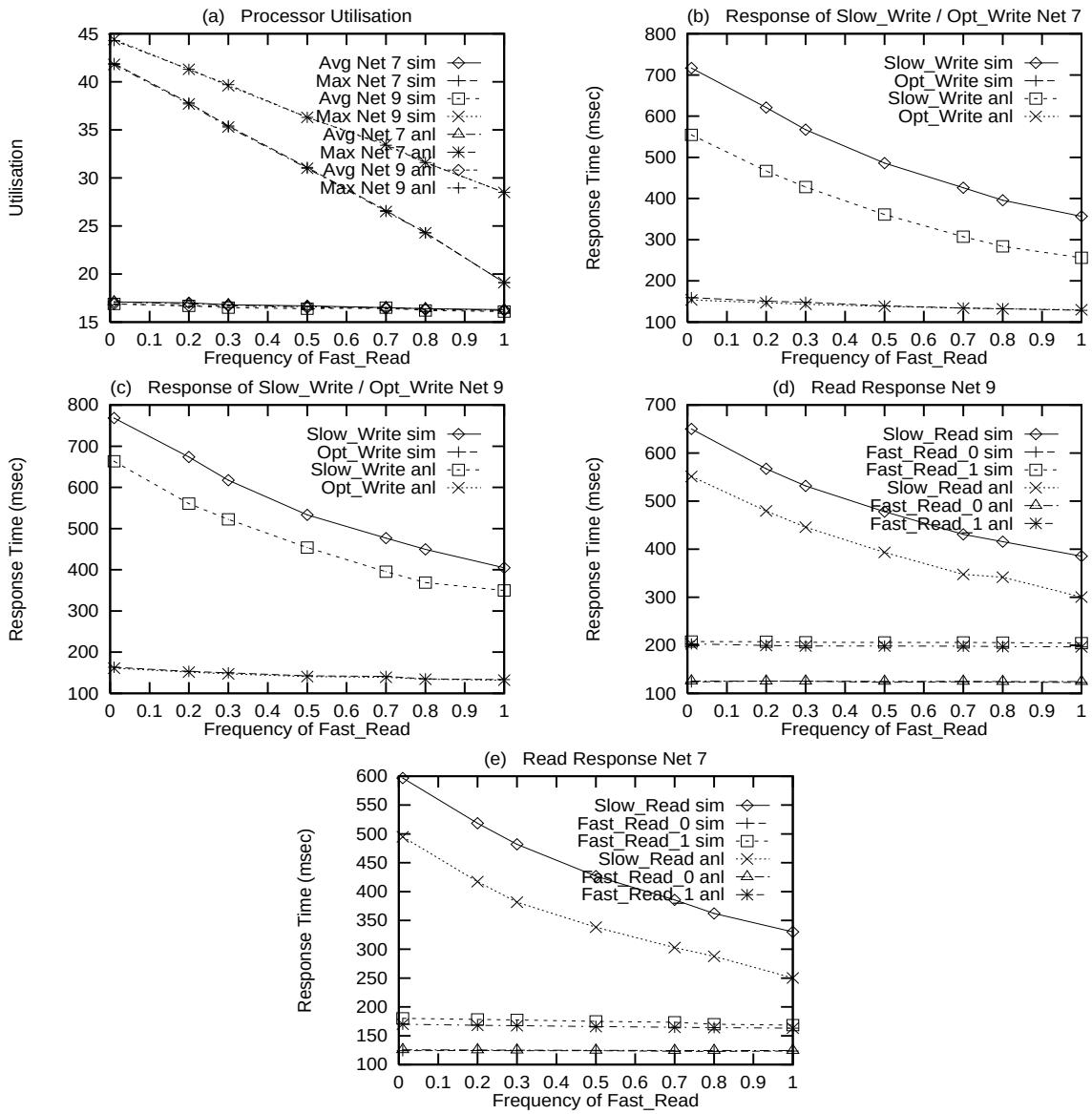


Figure 40: Varying read mix, Net 7 and Net 9

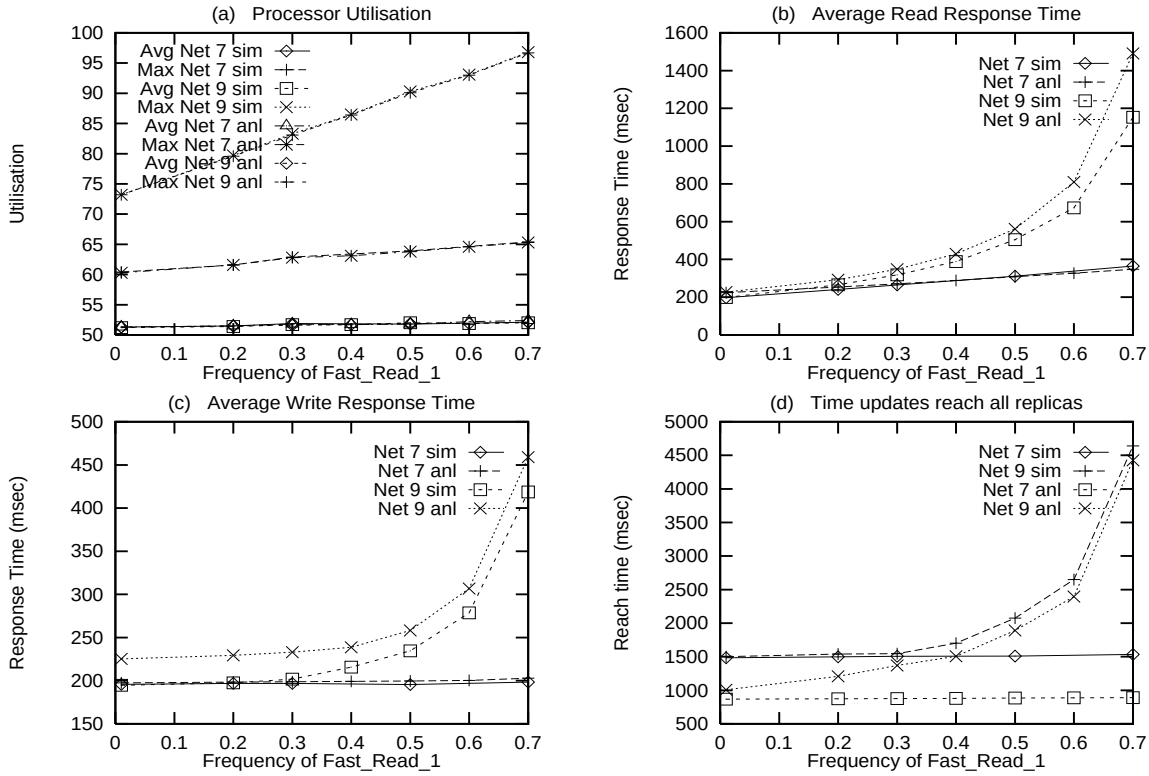


Figure 41: Varying read mix, Net 7 and Net 9, writes are Fast_Write

### 5.2.7 Varying read mix, comparing Net 7 and Net 9, writes are Fast_Writes

Varying the mix of Fast_Read_0 and Fast_Read_1 with writes being Fast_Write, we observe that utilisation of the resources are identical as shown in Figure 41 (a).

The response time of reads and writes ( Figure 41 (b) and (c)) are nearly identical analytically and in simulation for Net 7 while they are higher analytically for Net 9 and the gap gets higher as the frequency of Fast_Read_1 increases. This is because as $fast_{r1}$ increases, the resources are more loaded, due to the communication overhead, especially for Net 9. So, the increase in the estimated processing time becomes higher than the reduction in the estimated communication time resulting in an increase in the response time. However, the analytical curves follow the same shape as the simulation ones and the same conclusions regarding comparing the two networks are obtained.

In contrary, the reach time (Figure 41 (d)) is lower analytically, since it involves a fair amount of communication processing and processing propagation overhead which are estimated lower than in simulation. However, as $fast_{r1}$ increases, the gap gets smaller since the increase in the estimated processing time balances the reduction in the estimated communication time.

### 5.3 Discussion

It has been observed that the analytical results are generally close to the ones obtained through simulation. The utilisation of the resources are identical for both models. Although there is a difference between the response and reach times obtained from the analytical model and those obtained through the simulation, but the differences are small. Further, the differences are justified by the approximations used in the analytical model. Also, the same conclusions drawn from the simulation results can be deduced from the analytical ones. Therefore, it is believed that the results presented by the simulation are correct.

Although the analytical solution is much quicker than the simulation, but we have relied on simulation results in our performance evaluation since it does not include the approximations used in the analytical model. Further, there are measures that could not be measured analytically such as the age of the information read and the probability of reading out of date information.

## 6 Conclusions

Scalability and performance are two goals of a wide area distributed service. A hierarchical protocol for managing large scale replicated data was presented where each node has to communicate with only a few other nodes, typically ones in its physical proximity, thus reducing communications overhead, improving latency and making the scheme scalable. A new service interface is proposed that allows the application to select its required degree of consistency and to offer different levels of staleness by querying from different levels of the hierarchy.

A detailed simulation study was performed to evaluate the benefits and losses resulting from using synchronous versus asynchronous operations under different system configurations and load mixes. It has been shown that synchronous operations slow down the system operation and affects the performance tremendously. The response time deteriorates while resources are underutilised and throughput is limited due to data contention. Asynchronous operations have shown much better performance than synchronous operations and can cope with heavily loaded systems as well as with high communication overhead with a little loss in reading slightly out of date information.

When the applications cannot reconcile conflicting updates, then Slow_Write or Opt_Write must be used. It has been shown that Opt_Write offers better response time, coverage and local age than Slow_Write while it suffers an increase in the overall and abort age. Therefore, it is believed that a ratio of $slow_w$:$opt_w$ ranging between 4:6 up to 6:4 is a good compromise. This is provided that the application can guarantee that more than 80% of its Opt_Write will commit, so that throughput degradation is avoided. For applications that have high update rates, then Fast_Write is more suitable than Slow_Write, as long as the application can reconcile conflicting updates, since it offers much better performance and throughput and can cope better with heavily loaded systems and contended networks.

The results obtained about read operations have suggested that if the applications can accept observing slightly stale information then they should increase their frequency of Fast_Read especially under heavy load conditions, such as high arrival rate or heavily loaded networks, as this offers better performance as well as lower age. Further, using the principle of locality, HARP provides read operations that return less stale information with little increase in the response time. Fast_Read_1 offers better age and lower probability of returning old information

than Fast_Read_0 while Fast_Read_0 has better performance in terms of response time and load distribution. It is believed that an even mix of $fast_{r0}$ and $fast_{r1}$ is a good compromise that combines the two benefits.

The performance of the protocol has been evaluated on several network topologies of 12 nodes that reflect a hierarchical structure. It has been shown that a symmetric hierarchical structure, where the load is evenly distributed among the nodes on the backbone yields the best results. For completeness, we have evaluated the performance of a network that is more or less fully connected and all nodes are in one logical cluster. Applying synchronous operations to such a network represents the performance of traditional protocols such as the majority quorum consensus protocol. It has been shown that performance of HARP on hierarchical networks is much better than on such a network since the former limits quorum assembly to a small number of nodes. Further, asynchronous operations have shown poor performance on this network since it places extra load on the nodes: communication and propagation overhead.

Experiments were conducted with a larger number of nodes (39 nodes) and the same conclusions were obtained. Further, the results have suggested that it is better to keep the number of nodes in the top cluster small to reduce the overhead of quorum assembly while using synchronous operations and to reduce the communication and propagation overhead on those nodes while using asynchronous operations. Also, it has been shown that with a large number of nodes, it is important to consider the load placed on the nodes in the top cluster and try to spread it to lower levels, if possible, by increasing the number of levels in the hierarchy.

An analytical study has been carried out to evaluate the performance of the suggested model. The goal of the study is to verify the results of the simulator. An approximate solution based on the *Open Queueing Network Model with Multiple Job Classes* has been presented and its results has been compared to those obtained from the simulator.

### Acknowledgements

Noha Adly has been supported by an FCO grant from the British Council.

## References

[Adl93] N. Adly. HARP: a hierarchical asynchronous replication protocol for massively replicated data. Technical Report TR-310, Computer Laboratory, University of Cambridge, UK, August 1993.

[ANB93] N. Adly, M. Nagi, and J. Bacon. A hierarchical asynchronous replication protocol for large scale systems. In *Proceedings of the IEEE Workshop on Parallel and Distributed Systems*, pages 152–157, Princeton, NJ, October 1993.

[AR91] M. Ammar and G. Rouskas. On the performance of protocols for collecting responses over a multiple-access channel. In *Proceedings of INFOCOM-91*, pages 1490–1499, 1991.

[AS92] A. Agrawala and D. Sanghi. Networks dynamics: an experimental study of the internet. In *IEEE Proceedings of GLOBECOM'92*, pages 782–786, December 1992.

[BCMP75] F. Baskett, K. Chandy, R. Muntz, and F. Palacios. Open, closed and mixed networks of queues with different classes of customers. *Journal of the ACM*, (22):248–260, 1975.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[CAA90] S. Cheung, M. Ammar, and M. Ahamad. The Grid protocol: a high performance scheme for maintaining replicated data. In *Proceedings of the IEEE 6th Int. Conf. on Data Engineering*, pages 438–445, 1990.

[CDY90] B. Ciciani, D. Dias, and P. Yu. Analysis of replication in distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(2), June 1990.

[CK94] S. Cheung and A. Kumar. Efficient quorumcast routing algorithms. In *IEEE Proceedings of Infocom'94*, pages 840–847, June 1994.

[CL91] M. Carey and M. Livny. Conflict detection tradeoffs for replicate data. *ACM Transactions on Database Systems*, 16(4):703–746, December 1991.

[CLO87] M. Carey, B. Lindsay, and R. Obermarck. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Transactions on Software Engineering*, 13(12):1348–1363, December 1987.

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Symposium on Principles of Distributed Computing, ACM SIGACT-SIGOPS*, pages 1–12, 1987.

[DGP90] A. Downing, I. Greenberg, and J. Peha. OSCAR: an Architecture for Weak Consistency Replication. In *Proceedings of IEEE PARABASE-90*, pages 350–358, March 1990.

[Fer78] D. Ferrari. *Computer systems Performance Evaluation.* Prentice Hall, 1978.

[Fis92] P. Fishwick. Simpack: getting started with simulation programming in C and C++. In *Winter Simulation Conference*, pages 154–162, Arlington, VA, December 1992.

[GH85] D. Gross and C. Harris. *Fundamentals of queuing theory.* John Willey and Sons Inc, 1985.

[Gif79] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th Symposium on Operating System Principles*, December 1979.

[GL91] R. Golding and D. Long. Accessing replicated data in large scale distributed system. *International Journal in Computer Simulation*, 1(2), June 1991.

[GL92] R. Golding and D. Long. Quorum oriented multicast protocols for data replication. In *IEEE Proceedings of 8th Int. Conf. on Data Engineering*, June 1992.

[Gol92] R. Golding. *Weak-consistency group communication and membership.* PhD thesis, University of California, Santa Cruz, 1992.

[Gra88] J. Gray. The cost of messages. In *ACM Symposium on Principles of Distributed Computing*, pages 1–7, August 1988.

[Kel75] F. Kelly. Networks of queues with customers of different types. *Journal of applied probability*, pages 542–554, 1975.

[Kel76] F. Kelly. Networks of queues. *Journal of applied probability*, (8):416–432, 1976.

[Kum91] A. Kumar. Hierarchical quorum consensus: a new algorithm for managing replicated data. *IEEE transactions on Computers*, 40(9):996–1004, September 1991.

[Lam86] B. Lampson. Designing a global name service. In *Proceedings of the 5th Symposium on Principles of Distributed Computing, ACM SIGACT-SIGOPS*, pages 1–10, 1986.

[LLS90] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: exploiting the semantics of distributed services. In *Proceedings of the 9th ACM symposium on Principles of Distributed Computing*, pages 43–57, Quebec City, CA, August 1990.

[LZGS84] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative system performance: computer system analysis using queueing network models.* Prentice-Hall, 1984.

[PKL91] C. Pu, F. Korz, and R. Lehman. a measurement methodology for wide area networks. Technical Report CUCS-044-90, Columbia University, New York, NY 10027, March 1991.

[QP93] P. Queinnec and G. Padiou. Flight plan management in a distributed air traffic control system. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, Kawasaki, Japan, March 1993.

[Que82] Special issue on Solution Techniques for Queuing Networks. *Communication of the ACM*, February 1982.

[SBN84] M. Schroeder, A. Birrell, and R. Needham. Experience with Grapevine. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[WB84] G. Wuu and A. Bernstein. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the third ACM Symposium on Principles of Distributed Computing*, pages 233–242, August 1984.

[Zha94] Yongguang Zhang. *Communication Experiments for Distributed Transaction Processing – from LAN to WAN.* PhD thesis, Purdue University, December 1994.