ELEN 7040

University of the Witwatersrand, Johannesburg

Software Engineering Investigation Project

On the Effects of Partial Failure and Latency on Throughput and Data Consistency in Transactional-Messaging Systems

---

Author: Adrian Freemantle

Student Number: 1558184

13 November 2017

Abstract

Distributed systems have become popular as single-node solutions are no longer able to meet the processing and availability requirements of large-scale systems. Distributed systems partition and replicate functionality and data across multiple nodes to provide increased processing capacity and fault tolerance; however, this introduces challenges of latency and partial failure. Partial failure may cause messages to be lost, arrive out of order, be delivered multiple times, and experience unbounded delays. Message Queueing, when used with distributed transactions, is able to provide guaranteed message delivery with strong consistency guarantees; however distributed transactions incur coordination overhead which increases latency, resulting in a trade-off between consistency and throughput. The impact of latency and short-lived partial failures on the throughput and consistency of a transactional messaging system were evaluated experimentally; finding that strong consistency is achievable, but at the cost of an 80% reduction in message throughput.

Table of Contents

List of Figures

List of Tables

1 Introduction

Distributed systems [1] have become near ubiquitous, as single-node solutions are no longer able to meet the processing and availability requirements of large-scale systems [2] [3]. Distributed systems address these problems by partitioning and replicating functionality and data across multiple nodes to provide increased processing capacity and fault tolerance [4] [5] [6] [7] [8]. The advantages of distribution, however, introduce the challenges of latency, partial failure, and concurrency [9] [10] [11] [12] [13]. Designs which do not consider these factors may result in systems that exhibit poor performance, scalability and availability characteristics [14] [15].

> *Today, we see new design pressures foisted onto programmers that simply want to solve business problems. Their realities are taking them into a world of almost-infinite scaling and forcing them into design problems largely unrelated to the real business at hand [10].*

Designing *reliable, fault-tolerant applications in a distributed system is difficult* [16]. Issues of partial failure and concurrency have an impact on the system's consistency as messages may be lost, arrive out of order, or be delivered multiple times [17] [18]. Message queueing [19] with distributed transactions [20] *can help shield programmers from much of the complexity of distributed programming* [2]; it ensures exactly-once and in-order message delivery and provides strong consistency guarantees [21].

Despite the advantages of distributed transactions with message queueing, its use is contentious as *there is a widespread belief that the [performance] costs of these mechanisms are too high* [2]. Many claims regarding the performance costs or benefits are anecdotal and unsubstantiated [22] [23], making it difficult to objectively assess the core design trade-offs of consistency and throughput associated with transactional messaging [9] [4] [7] [11] [13] [24]. Failing to evaluate trade-offs during the design stage *may result in result in a system that is harder to implement, verify, understand, debug, and maintain* [15].

Section 2 reviews existing literature and presents the differences between local and distributed computing models. The issues of latency and partial failure are shown to result in a trade-off between consistency and throughput in distributed systems, with examples provided of how transactional messaging improves data consistency at the cost of reduced throughput.

Section 3 defines a research problem with hypotheses proposed on how latency and short-lived partial failures affect throughput and data consistency of transactional messaging. The research methodology used to evaluate the hypotheses is presented.

Section 4 presents the experiment design used to evaluated the proposed hypotheses. Dependent variables, independent variables, and steps taken to minimise the effects of confounding variables are discussed. Finally, Section 5 presents the experiment results and research findings.

2 Literature Review

Interest in distributed systems is increasing [2] [9] due to the processing capacity for eCommerce, financial trading, massively multiplayer online games, search engines, and Big-Data solutions *increasing at an exponential rate* [25]. Global computing capacity *is expected to be on the order of exascale computing by 2019* [25] with *an estimated 35 Zettabytes of data being stored by 2020*. The Internet of Things has resulted in billions of smart devices [3] [26] being connected to the Internet, each streaming sensor information for processing by so called Big Data [27] solutions. Availability and latency characteristics of these solutions have direct financial implications [7]. Amazon reported that *100ms of additional latency resulted in a 1% drop in sales* [7], with a 2014 survey by AppDynamics finding that a critical system outage for a Fortune 1000 company resulted in losses of up to $1 million per hour [28].

2.1 Communication Challenges of Distributed Computing

Failure of components within a distributed system is unavoidable [12] [18]. An average Google data centre experiences approximately 1,000 server failures per year [29], with power failure, damage to infrastructure, *network congestion, maintenance, overload, and errors due to operators* [30] further contributing to system downtime. Microsoft reported 13,300 network failures per year within a data centre [18], with HP finding that 11.4% of support tickets were due to network related problems [31].

The design of a distributed system must take these failures into account [14] [15] [18]. Failure of a component or process in local computing models is either detectable or results in the total failure of the application [12]. In distributed systems, independent components or processes may fail while others continue to operate correctly as there is no central coordinating process that can inform individual components of a failure [12] [32]. These partial failures may result in a network-partition, *a communication fault that splits the network into subsets of nodes such that nodes in one subset cannot communicate with nodes in another* [16]. Network partitions result in a state of uncertainty in which one node cannot make assumptions about the state of another, unresponsive node, due to a slow or failed node being indistinguishable from a failed network link [12] [16].

Peter Deutsch's *Fallacies of Distributed Computing* [5] [17] [18] lists some common assumptions that designers who are used to working with local system make when creating distributed systems:

- **The network is reliable**: Distributed applications communicate over networks in which individual components can fail due to overheating, misconfiguration, power failure, physical damage, network congestion, maintenance etc. [30]
- **Latency is zero**: The speed at which information can travel is limited by the speed of light [5], resulting in higher latency as the physical distance between nodes is increased. The number of sub-networks a network packet has to traverse increases latency due to processing overheads incurred by routers [5].
- **Bandwidth is infinite**: Networks have a limit to the amount of data that can be transmitted in a fixed amount of time, with network congestion occurring when the amount of data being transmitted approaches the available bandwidth limit [1].

- **The network is secure**: Distributed systems are susceptible a to interference by malicious actors [18]. For example, communication may be disrupted through Denial-of-Service attacks; physical infrastructure may be compromised, damaged or stolen [33]; and fraudulent nodes may attempt to *mislead or break computations* [14].

Deutsch's fallacies provide an indication of the kinds of challenges faced by distributed systems. Messages may be lost, arrive out of order, be delivered multiple times, and experience unbounded delays [17] [18]. Protocols such as TCP provide *exactly-once and in-order byte delivery between two communicating processes* [34] but provide no guarantees that the receiving process successfully processed the message; for example, the receiving process may be acting as a load-balancer which simply forwards the message to another node for processing, or the receiver may unexpectedly terminate before the message was fully processed [34]. Furthermore, TCP temporally couples [35] the sender to receiver, as the receiver must be active and able to receive the message when it is sent [34].

2.2 Message Queueing

One solution to the communication challenges faced by distributed systems is to use Message Queuing to enable processes to communicate in an asynchronous and temporally decoupled manner. Microsoft Messaging Queuing (MSMQ) [36] *provides guaranteed message delivery, efficient routing, security, transaction support, and priority-based messaging* [21]. Message queues, which act as durable message stores [37], are used to send and receive messages while the MSMQ infrastructure ensures that messages are successfully routed between queues [36]. For instance, the application shown in Figure 1 receives a message by dequeuing it from its incoming message queue, processes the message, updates its database, and sends a new message by enqueueing it the outgoing message queue.

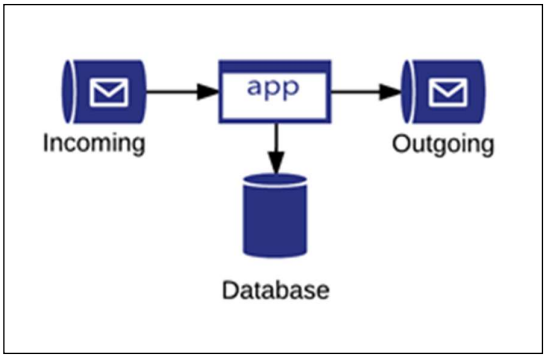

Figure 1: Application Integration with Message Queues

2.3 Transactions

A core problem faced in the design of distributed systems is ensuring that the overall system state is consistent after a network partition or partial-failure [12]. Local transactions are able to exhibit ACID properties; Atomic, Consistent, Isolated and Durable [1]. Atomic consistency guarantees ensure that that all actions performed within the scope of the transaction appear to have been performed at a single instant in time or fail in their

entirety [24]. ACID guarantees are only valid for data contained within a single partition [5] [9] [24]; as a result, the application database and MSMQ infrastructure each have separate local transactions.

2.3.1 Distributed Transactions

Atomic consistency is not possible with separate transactions as each transaction can fail or succeed independently [5] [24]; however, this problem can be partially mitigated by using an atomic commit protocol [1] to coordinate multiple transactions.

> *An atomic commit protocol is a cooperative procedure used by a set of servers involved in a distributed transaction. It enables the servers to reach a joint decision as to whether a transaction can be committed or aborted* [1].

An example of an atomic commit protocol is the Two-Phase Commit [20] (2PC) protocol which is used by the Microsoft Distributed Transaction Coordinator [38] (MS-DTC). 2PC uses a single transaction manager (TM) to coordinate the actions of multiple resource managers (RM), with each RM managing individual transactions. The TM will issue a commit instruction to each RM if all transactions are able to commit, or issue a rollback if any RM is not able to commit its transaction [20].

2.3.2 Coordination Costs

Protocols like 2PC provide a façade of atomic consistency at the cost of decreased throughput due to coordination costs incurred between the TM and RMs. As shown in Figure 2, a distributed transaction involving an application RM, database RM, and MSMSQ RM incurs a coordination cost of five additional messages.

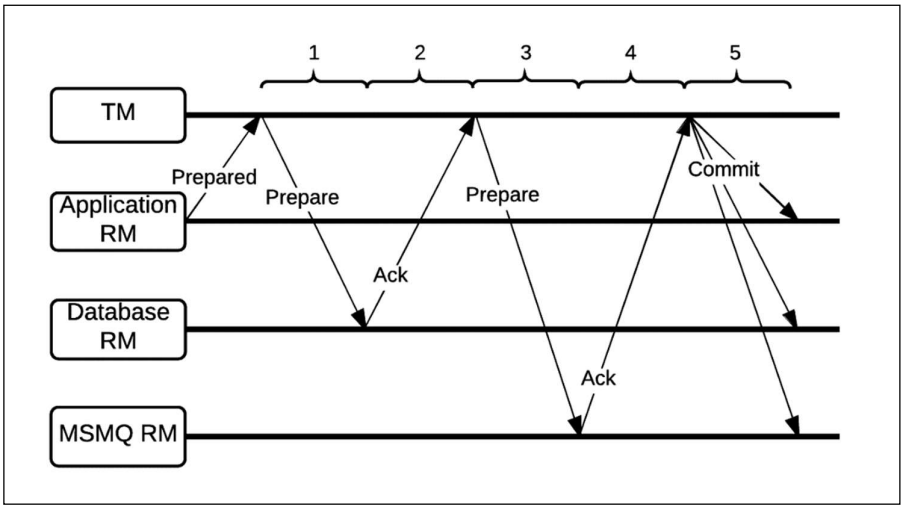

Figure 2: Distributed Transaction Coordination Costs [20]

2.4 Transactional Messaging

Just as Relational Database Management Systems provide different transaction isolation levels to provide different levels of concurrency [16], MSMQ supports various

transactional-messaging models to provide different consistency guarantees with different throughput characteristics [36]:

- **Non-transactional (N-TX):** Messages are permanently removed from the queue when dequeued and are immediately sent when enqueued. This provides at-most-once delivery and zero-or-once processing guarantees.
- **Queue transactions (Q-TX):** Message queue operations (enqueuing and dequeuing) are atomic within the scope of a local message queue transaction. Enqueued messages are sent and dequeued messages are removed once the transaction has been committed. This provides at-least-once delivery and at-least-once processing guarantees.
- **Distributed transactions (D-TX):** The Microsoft Distributed Transaction Coordinator (MS-DTC) is used to provide atomic consistency guarantees between message queues and the application database. This provides exactly-once delivery and exactly-once processing guarantees.

2.4.1 Consistency Guarantees

This section provides some examples of how the same physical design experiences different data consistency guarantees based on the transactional messaging model used.

The distributed system in Figure 3 provides an example of two nodes which communicate asynchronously through messaging queueing using a request-response communication pattern [19]. The process is initiated when a client application enqueues a message for Node-A to transfer R100 from an account located at Node-A to an account located at Node-B. The transfer is considered complete once Node-A receives a confirmation message from Node-B.

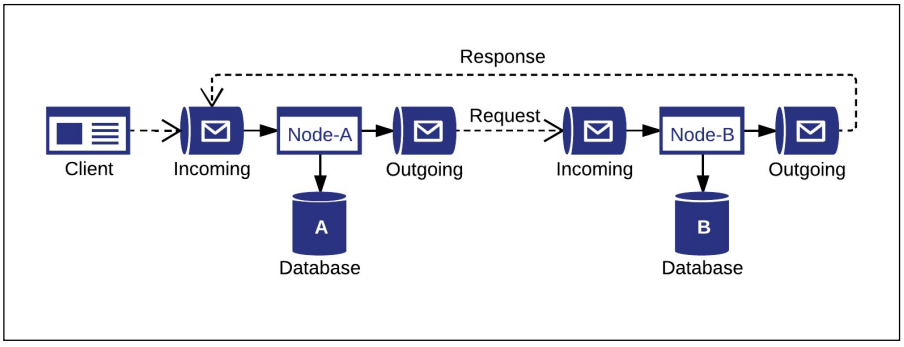

Figure 3: Communication between Two Nodes Using Message Queueing

2.4.1.1 D-TX: Exactly-Once Message Processing

Node-A dequeues the message, debits R100.00 from the source account, marks the transfer as pending, and enqueues a message which instructs Node-B to credit the destination account with R100.00. Node-B dequeues the message, credits the destination account with R100.00, and sends a response message to Node-A to acknowledge that the transfer has completed. Node-A receives the response and marks the transfer as completed. The system's state is consistent at the end of this process.

2.4.1.2 Q-TX: At-Least-Once Message Processing

Node-B starts a message queue transaction, dequeues the message, credits the destination account, and enqueues a response message. The node fails after it commits the database transaction but before it commits the message queue transaction, resulting in the dequeued message being returned to the incoming queue and the enqueued response being removed from the outgoing queue. The message is considered undelivered and is dequeued again when the node recovers, effectively resulting in multiple credits to the destination account.

2.4.1.3 N-TX: At-Most-Once Message Processing

Node-B dequeues the message but crashes before it fails to process it, resulting in the funds not being credited to the destination account and no response message being sent. Alternately, Node-B processes the message and credits the destination account but fails before it is able to enqueue a response. Node-A has no way of knowing whether the lack of a response indicates that Node-B failed to process the message, failed to enqueue a response, or whether Node-B is temporarily unavailable or very slow. Node-A could use timeouts and retries to resend the transfer instruction; however, this could result in Node-B processing the instruction multiple times, effectively resulting in multiple credits to the destination account.

3 Research Problem

As mentioned in Section 1, many claims regarding the performance costs of distributed transactions with message queueing are anecdotal and unsubstantiated. This presents a challenge when attempting to objectively assess the design trade-offs associated with different transactional messaging models.

3.1 Question

What is the impact of latency and short-lived partial failures on the throughput and data consistency characteristics of a distributed system using transactional messaging with MSMQ?

3.2 Hypotheses

3.2.1 Message Throughput

- **Null Hypothesis:** Latency and packet loss have no impact on the message throughput of transactional messaging when compared to non-transactional messaging.
- **Hypothesis:** Latency and packet loss have an adverse impact on the message throughput of transactional messaging when compared to non-transactional messaging.

3.2.2 Data Consistency

- **Null Hypothesis:** Messaging with distributed transactions does not provide atomic consistency guarantees between the message queue and the application database when faced with short-lived partial failures.

- **Hypothesis:** Messaging with distributed transactions provides atomic consistency guarantees between the message queue and the application database when faced with short-lived partial failures.

3.3 Methodology

An experimental research design was used in which quantitative and independent variables were manipulated to obtain results which could be statistically analysed to prove or disprove the hypotheses in Section 3.2.

4 Experimental Design

The aim of the experiment was to determine the impact of latency and short-lived partial failures on the throughput and consistency characteristics of a distributed system using transactional messaging with MSMQ. The system represented by Figure 4 was used to process messages sent by a client through two distributed nodes. Consistency was measured by comparing the number of messages sent by the client to the number of duplicate or missing records in the database of Node-B. Throughput was measured as messages-per-second processed by the entire system.

Three identical virtual-machines were used to host Node-A, Node-B and the Microsoft SQL Server instance. The virtual machines' network adapters were bridged with the host system's network adapter to create a virtual local-network which would exhibit near zero latency and experience zero packet loss. Hardware and software configurations of each virtual-machine and the host system are provided in Appendix C – Experiment Hardware Configuration.

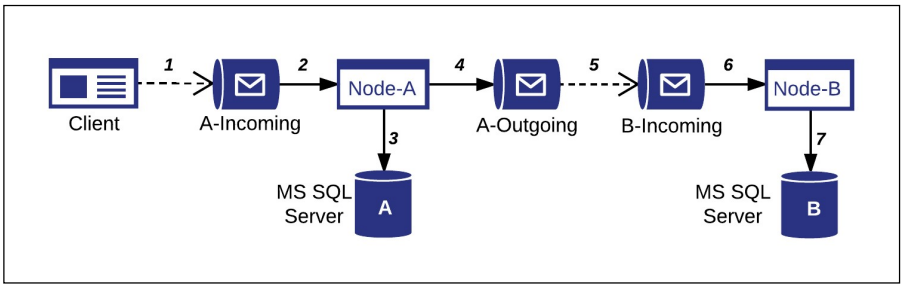

Figure 4: Test Configuration

The process for each message is:

1. A client application sends a message containing a Globally Unique Identifier (GUID) to the incoming queue of Node-A.
2. Node-A dequeues a message from its incoming queue.
3. Node-A inserts a database record containing the GUID from the message.
4. Node-A enqueues the message to its outgoing queue with Node-B set as the destination.
5. MSMQ ensures that the message is delivered to the incoming queue of Node-B.
6. Node-B dequeues a message from its incoming queue.
7. Node-B inserts a database record containing the GUID from the message.

4.1 Experiments

Short-lived network partitions were modelled using a *fail-fast* [39] approach in *which a component is either functioning correctly or simply stops functioning* [39]. Configurable failure points were introduced to independently simulate short-lived failure of either the messaging infrastructure or the application database.

Latency and packet loss were simulated using Clumsy [40], an application which intercepts network packets being received or sent by the operating system. Clumsy can be configured to intercept a specified percentage of packets to simulate packet loss or simulate latency by delaying packets.

The dependent and independent variables for each experiment are listed in Table 1:

Table 1: Dependent and Independent Variables

Experiment No.	Dependent Variable	Independent Variable
1.	0.10% Error after Enqueue	Consistency
2.	0.10% Error after Dequeue	Consistency
3.	0.10% Error after DB TX Commit	Consistency
4.	50ms Latency	Throughput
5.	5.00% Packet Loss	Consistency Throughput

Each experiment, and control, was executed 10 times against each transactional messaging model, with 10,000 messages per execution. Multiple executions per experiment were used to average out the impact of confounding variables such as garbage collection [41] and background operating-system processes.

5 Research Findings

This section presents the summarised results of the experiments outlined in Section 4, the detailed results are available in Appendix B – Detailed Experiment Results.

5.1 Message Throughput

The median message throughput for the control, latency and packet loss tests are shown in Figure 5**Error! Reference source not found.**. In the control test, N-TX was found to have the highest throughput at 178.57 messages per second, followed by Q-TX at 61.35 messages per second, and finally D-TX at 30.12 messages per second.

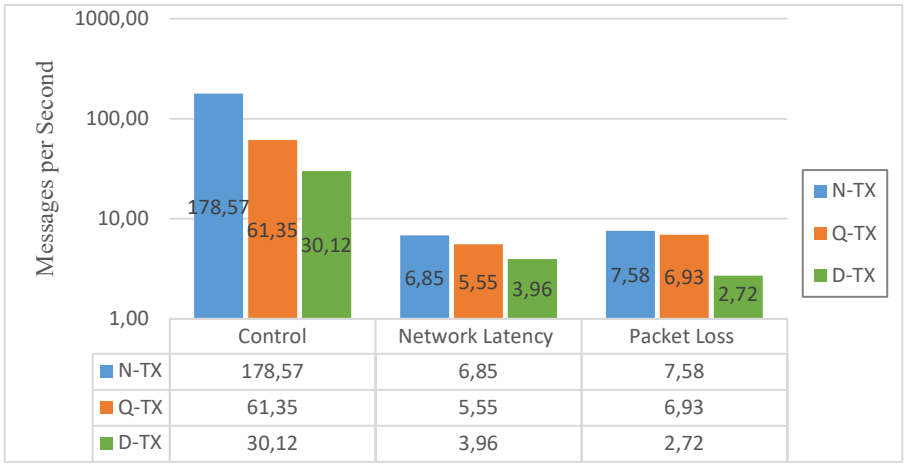

Figure 5: Median Messages-per-Second

As shown in Figure 6, the relative difference in throughput for Q-TX compared to N-TX was only reduced by 19.1% for latency experiments, and 8.6% percent for packet loss experiments; significantly less than the 66.6% reduction encountered in the control. Similarly, the relative difference in throughput for D-TX compared to N-TX improved in latency and packet loss experiments when compared to the control.

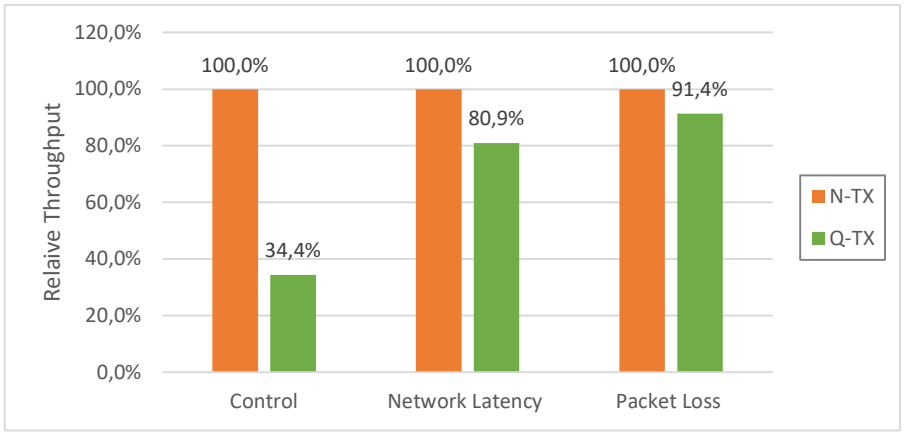

Figure 6: Relative Throughput of Q-TX and D-TX Compared to N-TX

The hypothesis from Section 3.2.1 is shown to be true; latency and packet loss have an adverse impact on the message throughput of transactional messaging when compared to non-transactional messaging.

5.2 Data Consistency

The number of duplicate or lost messages for the control and data consistency tests are shown in Figure 7. N-TX resulted in missing data records when simulating MSMQ failures and packet loss. Q-TX resulted in duplicate data records when simulating packet loss, and errors occurring after the database transaction commit but before the MSMQ transaction commit. D-TX resulted in perfect data consistency in all tests.

Figure 7: Duplicate or Lost Records

The hypothesis from Section 3.2.2 is shown to be true; D-TX provides atomic consistency guarantees between the MSMQ and the application database when faced with short-lived partial failures.

6 Conclusion

The impact of latency and short-lived partial failures on the throughput and consistency characteristics of a distributed system using transactional messaging with MSMQ were evaluated experimentally. Results were found to be consistent with anecdotal claims; Stronger consistency guarantees were shown to have a correlated reduction in throughput.

Non-transactional messaging was shown to provide the fastest throughput but resulted in data inconsistency in the form of missing records. Transactional messaging was shown to be three times slower than non-transactional messaging, however, multiple processing of some messages resulted in data inconsistency in the form of duplicate records. Messaging with distributed transactions exhibited perfect data consistency but was shown to be two to five times slower than non-transactional messaging.

6.1 Future Work

Messaging with distributed transactions is easier for developers to work with as it provides a façade of ACID transactions, however this comes at the cost of reduced concurrency and limits the ability of a system to effectively scale horizontally [2] [5]. For systems that must achieve near infinite scalability, the cost of strong consistency is considered too high [14] [6]. These systems prefer weaker consistency models which provide increased throughput and lower latency responses by eliminating the coordination overhead required by atomic commit protocols [2] [5] [32]. Further investigation is required to determine the advantages and disadvantages of weaker consistency models and identify ways to handle inconsistencies.

References

[1] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, Distributed Systems: Concepts and Design, Boston, Massachusetts: Addison-Wesley, 2012.

[2] P. Alvaro, N. Conway, J. M. Hellerstein and W. R. Marczak, "Consistency Analysis in Bloom: a CALM and Collected Approach," *CIDR,* pp. 249-260, 2011.

[3] D. Evans, "Cisco Consulting Thought Leadership," April 2011. [Online]. Available: http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf. [Accessed 15 May 2015].

[4] A. Fox and E. A. Brewer, "Harvest, yield, and scalable tolerant systems.," in *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, 1999.

[5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein and I. Stoica, "Highly Available Transactions: Virtues and Limitations," *Proceedings of the VLDB Endowment,* vol. 7, no. 3, pp. 181-192, 2013.

[6] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR),* vol. 22, no. 4, pp. 299-319, 1990.

[7] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein and I. Stoica, "Quantifying eventual consistency with PBS," *The VLDB Journal,* vol. 23, no. 2, pp. 279-302, 2014.

[8] H. Garcia-Molina and K. Salem, "Sagas," ACM, 1987.

[9] G. Ramalingam and K. Vaswani, "Idempotence, Fault Tolerance via Idempotence," *ACM SIGPLAN Notices,* vol. 48, no. 1, pp. 249-262, 2013.

[10] P. Helland, "Life beyond Distributed Transactions: an Apostate's Opinion.," *CIDR,* vol. 2007, pp. 132-141, 2016.

[11] P. Mahajan, L. Alvisi and M. Dahlin, "Consistency, availability, and convergence," *University of Texas at Austin Tech Report,* vol. 11, 2011.

[12] J. Waldo, G. Wyant, A. Wollrath and S. Kendall, "A note on distributed computing.," *Mobile Object Systems Towards the Programmable Internet,* pp. 49-64, 1997.

[13] J. M. Carlson and J. Doyle, "Highly Optimized Tolerance: A Mechanism for Power Laws in Designed Systems," *Physical Review E,* vol. 60, no. 2, p. 1412, 1999.

[14] Y. Brun and N. Medvidovic, "Fault and adversary tolerance as an emergent property of distributed systems' software architectures," in *ACM*, 2007.

[15] E. A. Akkoyunlu, K. Ekanadham and R. Huber, "Some constraints and tradeoffs in the design of network communications," *ACM SIGOPS Operating Systems Review,* vol. 9, no. 5, pp. 67-74, 1975.

[16] M. Kleppmann, "A Critique of the CAP Theorem," 2015.

[17] J. Sheehy, “There is no now,” *Communications of the ACM,* vol. 58, no. 5, pp. 36-41, 2015.

[18] P. Bailis and K. Kingsbury, “The Network is Reliable,” *Queue,* vol. 12, no. 7, pp. 20-32, July 2014.

[19] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2004.

[20] J. Gray and L. Lamport, “Consensus on transaction commit,” *ACM Transactions on Database Systems (TODS),* vol. 31, no. 1, pp. 133-160, 2006.

[21] Microsoft, “Message Queuing Overview,” Microsoft, 19 July 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms703216(v=vs.85).aspx. [Accessed 17 October 2017].

[22] J. Oliver, “My Beef with MSDTC and Two-Phase Commits,” Jonathan Oliver, 4 April 2011. [Online]. Available: http://blog.jonathanoliver.com/my-beef-with-msdtc-and-two-phase-commits/. [Accessed 16 06 2017].

[23] G. Young, “Idempotency vs Distibuted Transactions,” CodeBetter.com, 12 August 2010. [Online]. Available: http://codebetter.com/gregyoung/2010/08/12/idempotency-vs-distibuted-transactions/. [Accessed 12 November 2017].

[24] S. Gilbert and N. Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,” *SIGACT News,* vol. 33, no. 2, pp. 51-59, June 2002.

[25] D. Patel, F. Khasib, I. Sadooghi and I. Raicu, “Towards in-order and exactly-once delivery using hierarchical distributed message queues.,” *Cluster, Cloud and Grid Computing,* vol. 2014, pp. 883--892, 2014.

[26] Freescale Semiconductor, Inc., “Freescale,” July 2014. [Online]. Available: http://www.freescale.com/files/32bit/doc/white_paper/INTOTHNGSWP.pdf. [Accessed 15 May 2015].

[27] P. Zikopoulos and et al., Harness the Power of Big Data, New York: McGraw-Hill, 2013.

[28] S. Elliot, “DevOps and the cost of downtime: Fortune 1000 best practice metrics quantified,” December 2014. [Online]. Available: https://kapost-files-prod.s3.amazonaws.com/published/54ef73ef2592468e25000438/idc-devops-and-the-cost-of-downtime-fortune-1000-best-practice-metrics-quantified.pdf. [Accessed 18 October 2017].

[29] J. Dean, “Design Lessons and Advice from Building Large Scale,” 2009. [Online]. Available: http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf. [Accessed 13 08 2017].

[30] M. Burrows, “The Chubby lock service for loosely-coupled distributed systems,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, Washington, 2006.

[31] D. Turner, K. Levchenko, J. Mogul, S. Savage and A. Snoeren, “On Failure in Managed Enterprise Networks,” HP Laboratories, 2012.

[32] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein and I. Stoica, "Coordination Avoidance in Database Systems," *Proceedings of the VLDB Endowment,* vol. 8, pp. 185-196, 2014.

[33] E. Alsaadi and A. Tubaishat, "Internet of Things: Features, Challenges, and Vulnerabilities," *International Journal of Advanced Computer Science and Information Technology,* vol. 4, no. 1, pp. 1-13, 2015.

[34] P. Helland, "Idempotence is not a medical condition," *Communications of the ACM,* vol. 55, no. 5, pp. 56-65, 2012.

[35] F. Beck and S. Diehl, "On the Congruence of Modularity and Code Coupling," *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering,* pp. 354-364, 2011.

[36] Microsoft, "Message Queuing (MSMQ)," Microsoft, 19 July 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms711472(v=vs.85).aspx. [Accessed 04 03 2017].

[37] R. Daigneau, Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, Addison-Wesley, 2012.

[38] Microsoft, "Distributed Transaction Coordinator," Distributed Transaction Coordinator, 19 July 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms684146.aspx. [Accessed 17 October 2017].

[39] P. Helland and D. Campbell, "Building on quicksand," *arXiv preprint arXiv:0909.1788,* 2009.

[40] "Clumsy 0.2," [Online]. Available: https://jagt.github.io/clumsy/. [Accessed 13 November 2017].

[41] M. Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2003.

Appendix A – Experiment Configurations

This section presents the test configurations for non-transactional, transactional and distributed transaction tests.

i) Non-Transactional Configuration

The Sequence Diagram in Figure 8 shows the interaction between the main components involved in the test. The Message Receiver polls the Incoming Queue for any received messages, with the *Dequeue* method removing a message from the Message Queue and returning it.

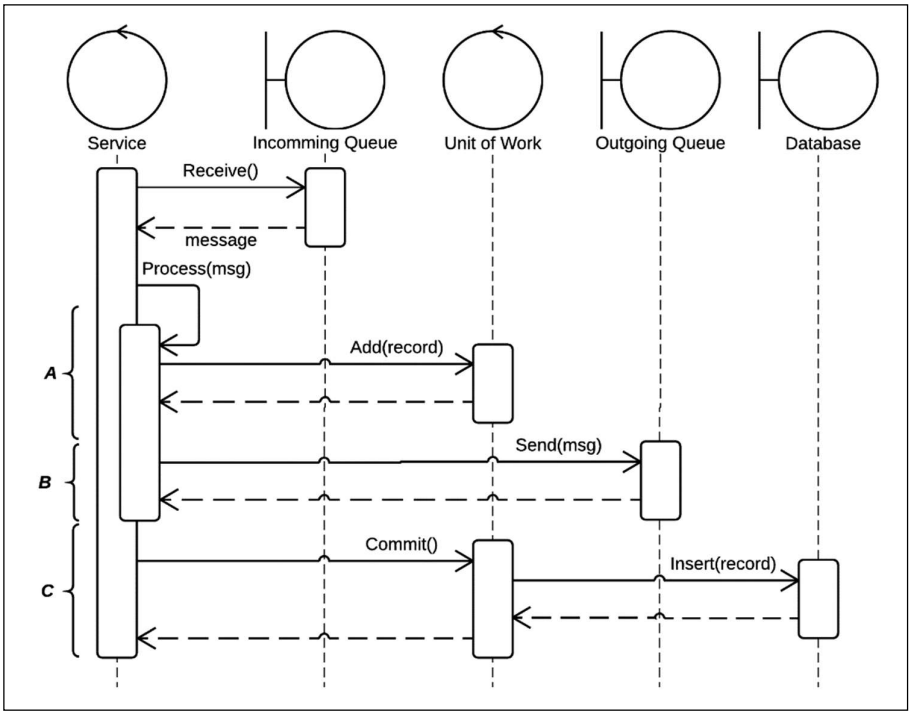

Figure 8: Simplified Sequence Diagram for Non-Transactional Tests

A. The Service receives a message from its incoming queue.
B. The received message is processed and the GUID contained in the message is used to create a new record which is added to the Unit-of-Work.
C. The received message is forwarded to the next service by placing it in an outgoing message-queue, providing the receivers address as a parameter.
D. The record created in step B is added to the database by committing the Unit-of-Work.

ii) Transactional Configuration

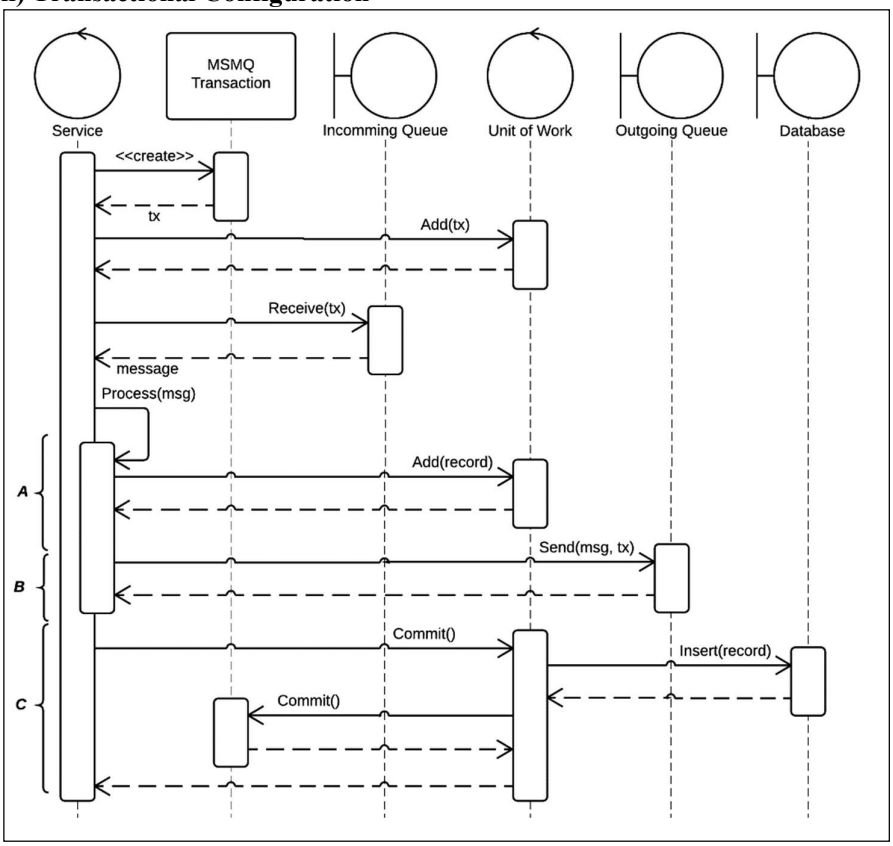

Figure 9: Simplified Sequence Diagram for Transactional Queue Tests

A. The Service creates an MSMQ Transaction object which is added to the Unit-of-Work and is passed to the incoming queue when receiving a message.
B. The received message is processed and the GUID contained in the message is used to create a new record which is added to the Unit-of-Work.
C. The received message is forwarded to the next service by placing it in an outgoing message-queue, providing the receivers address and MSMQ Transaction object as parameters.
D. The record created in step B is added to the database by committing the Unit-of-Work. The Unit-of-Work also commits the MSMQ Transaction, resulting in the outgoing message being sent and the received message being deleted from the incoming queue.

iii) Distributed-Transactional Configuration

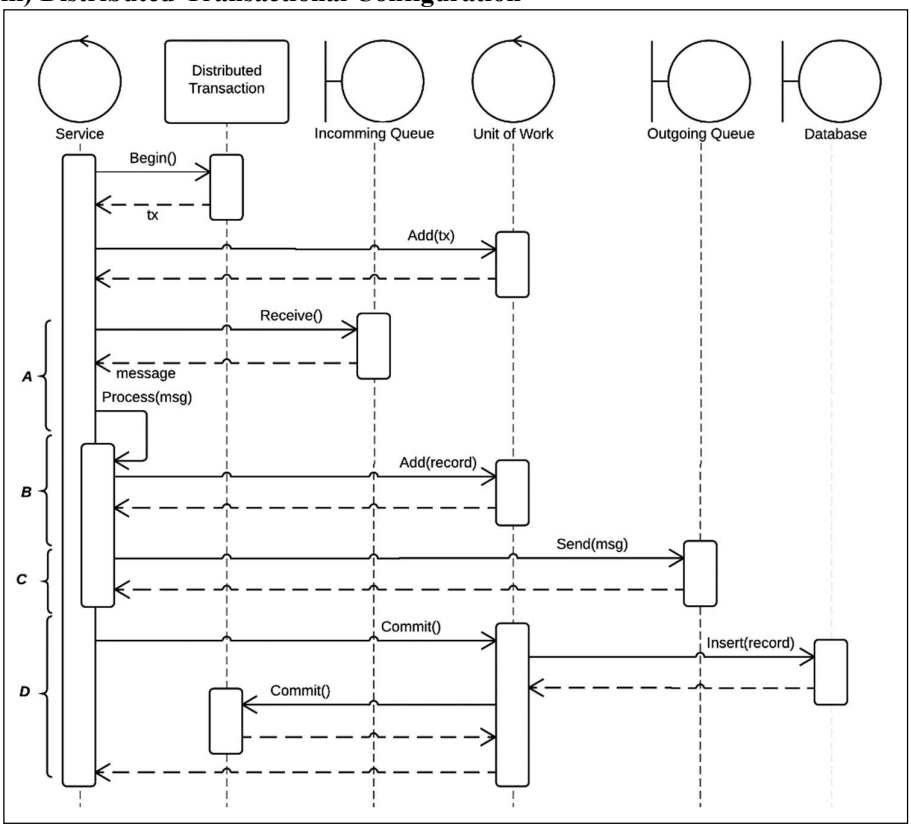

Figure 10: Simplified Sequence Diagram for Distributed Transaction Tests

A. The Service creates a Distributed Transaction object which is added to the Unit-of-Work. The incoming message queue is automatically enlisted into the ambient Distributed Transaction when a message is retrieved from it.
B. The received message is processed and the GUID contained in the message is used to create a new record which is added to the Unit-of-Work.
C. The received message is forwarded to the next service by placing it in an outgoing message-queue, providing the receivers address as a parameter. The outgoing message queue is automatically enlisted into the ambient Distributed Transaction
D. The record created in step B is added to the database by committing the Unit-of-Work, with the database transaction automatically being enlisted into the ambient Distributed Transaction. All transactions are resolved when the Unit-of-Work commits the Distributed Transaction, resulting in the outgoing message being sent, the received message being deleted from the incoming queue, and the database transaction being committed.

Appendix B – Detailed Experiment Results

i) Control

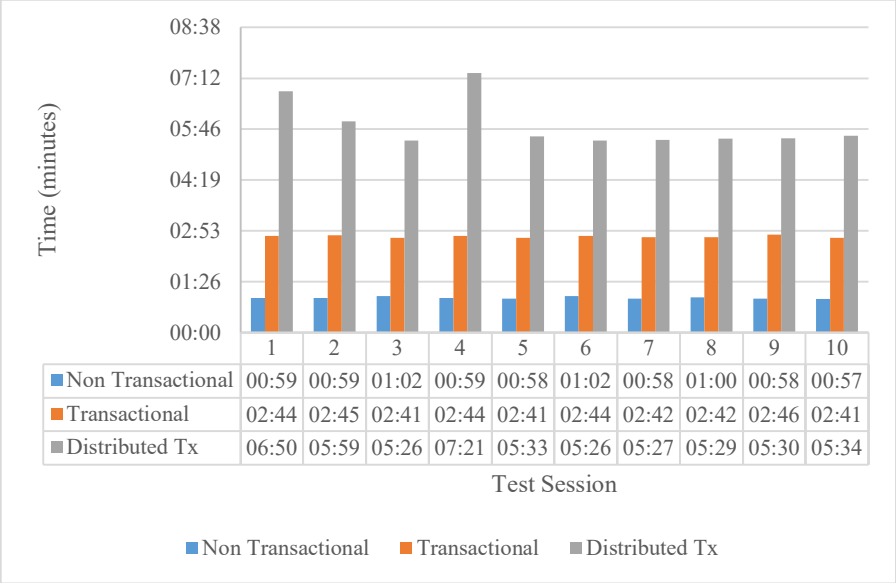

Figure 11:Completion Times for Control Tests

Table 2: Completion Time Summary for Control Tests

Transaction	Minimum	Maximum	Average	Median
None	00:00:56	00:01:02	00:00:59	00:00:59
Transactional	00:02:41	00:02:46	00:02:43	00:02:43
Distributed Transaction	00:05:26	00:07:21	00:05:51	00:05:32

Table 3: Duplicate and Lost Record Summary for Control Tests

Transaction	Node-A Delta	Node-B Delta	Total Delta
None	0	0	0
Transactional	0	0	0
Distributed Transaction	0	0	0

ii) Error after Message Dequeue

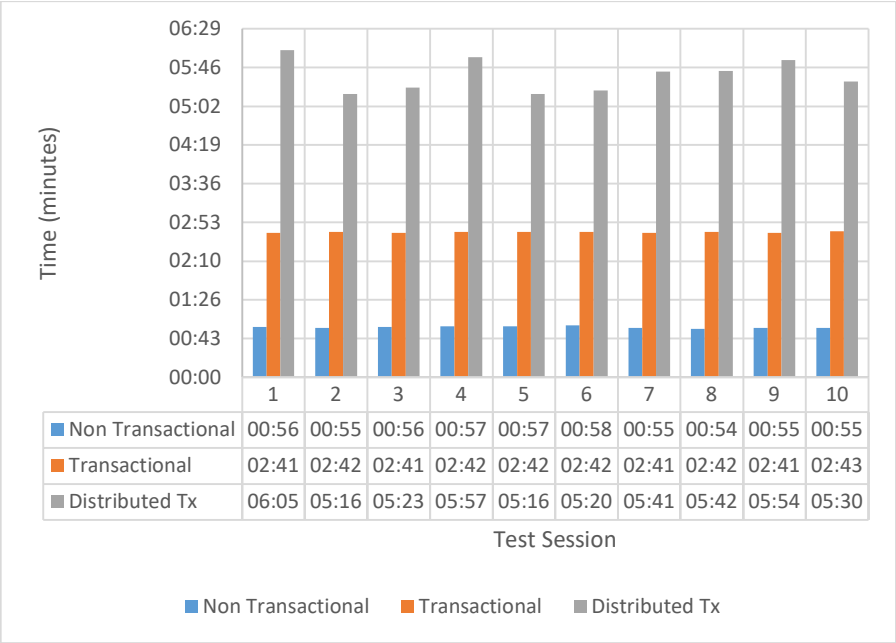

Figure 12: Completion Times for Error after Dequeue Tests

Table 4: Completion Time Summary for Error after Dequeue Tests

Transaction	Minimum	Maximum	Average	Median
None	00:00:54	00:00:58	00:00:56	00:00:56
Transactional	00:02:41	00:02:43	00:02:42	00:02:42
Distributed Transaction	00:05:16	00:06:05	00:05:36	00:05:36

Table 5: Duplicate and Lost Record Summary for Error after Dequeue Tests

Transaction	Node-A Delta	Node-B Delta	Total Delta
None	-88	-88	-176
Transactional	0	0	0
Distributed Transaction	0	0	0

iii) Error after Message Enqueue

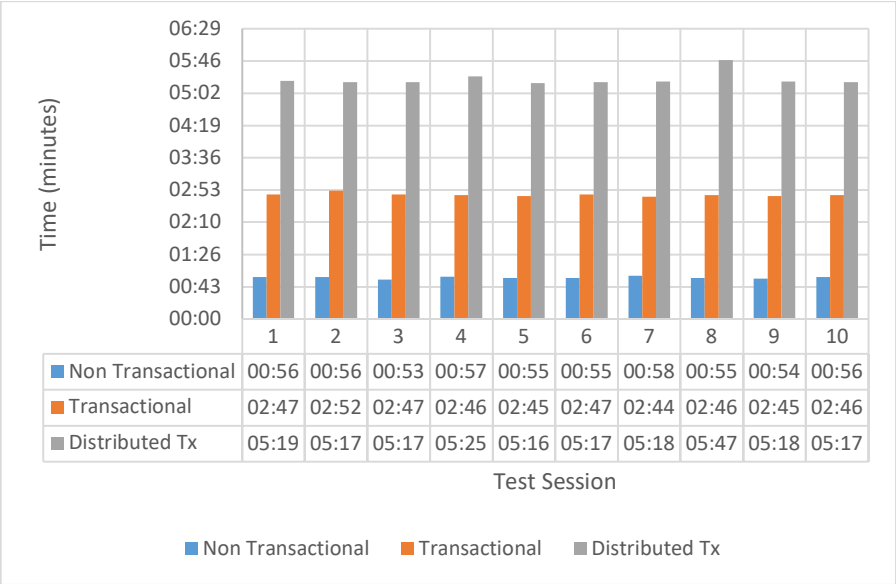

Figure 13: Completion Times for Error after Enqueue Tests

Table 6: Completion Time Summary for Error after Enqueue Tests

Transaction	Minimum	Maximum	Average	Median
None	00:00:54	00:00:58	00:00:56	00:00:56
Transactional	00:02:41	00:02:43	00:02:42	00:02:42
Distributed Transaction	00:05:16	00:06:05	00:05:36	00:05:36

Table 7: Duplicate and Lost Record Summary for Error after Enqueue Tests

Transaction	Node-A Delta	Node-B Delta	Total Delta
None	-103	0	-103
Transactional	0	0	0
Distributed Transaction	0	0	0

iv) Error after Database Commit

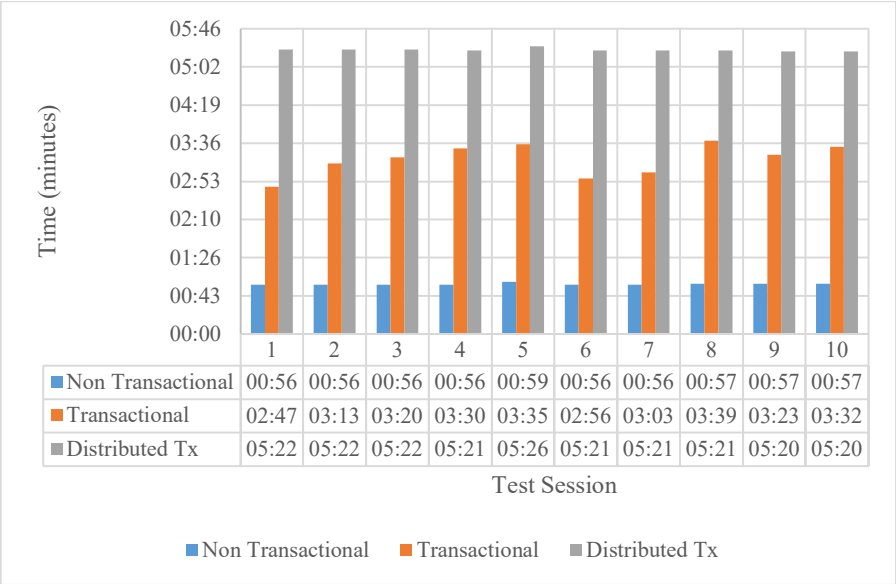

Figure 14: Completion Times for Error after Database Commit Tests

Table 8: Completion Time Summary for Error after Database Commit Tests

Transaction	Minimum	Maximum	Average	Median
None	00:00:56	00:00:59	00:00:57	00:00:56
Transactional	00:02:47	00:03:39	00:03:18	00:03:22
Distributed Transaction	00:05:20	00:05:26	00:05:22	00:05:21

Table 9: Duplicate and Lost Record Summary for Error after Database Commit Tests

Transaction	Node-A Delta	Node-B Delta	Total Delta
None	0	0	0
Transactional	98	107	205
Distributed Transaction	0	0	0

v) Network Packet Loss

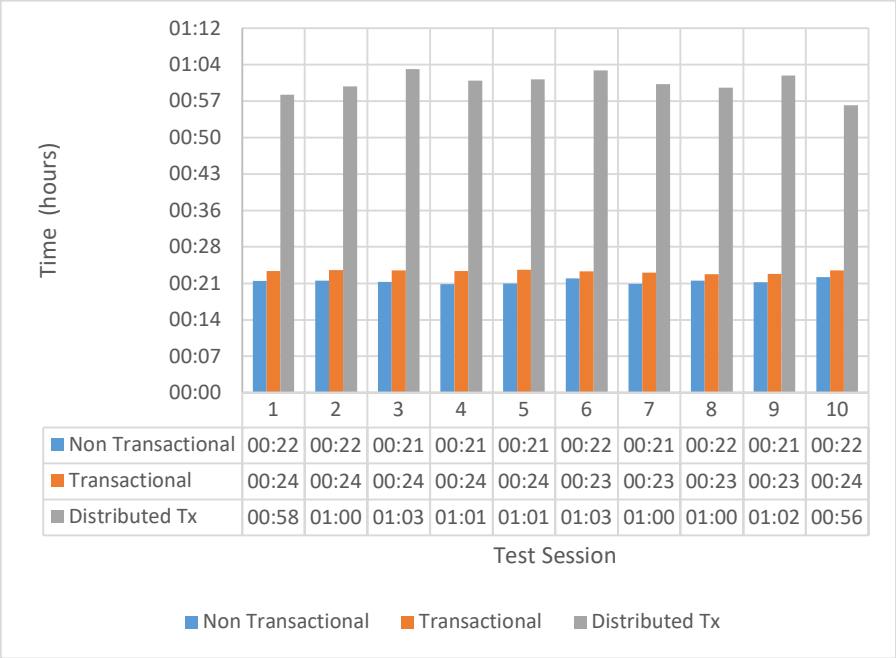

Figure 15: Completion Times for Packet Loss Tests

Table 10: Completion Time Summary for Packet Loss Tests

Transaction	Minimum	Maximum	Average	Median
None	00:21:27	00:22:51	00:22:00	00:22:00
Transactional	00:23:26	00:24:17	00:23:58	00:24:04
Distributed Transaction	00:56:49	01:03:55	01:01:08	01:01:21

Table 11: Duplicate and Lost Record Summary for Packet Loss Tests

Transaction	Node-A Delta	Node-B Delta	Total Delta
None	-4	-1	-5
Transactional	0	1	1
Distributed Transaction	0	0	0

vi) Network Latency

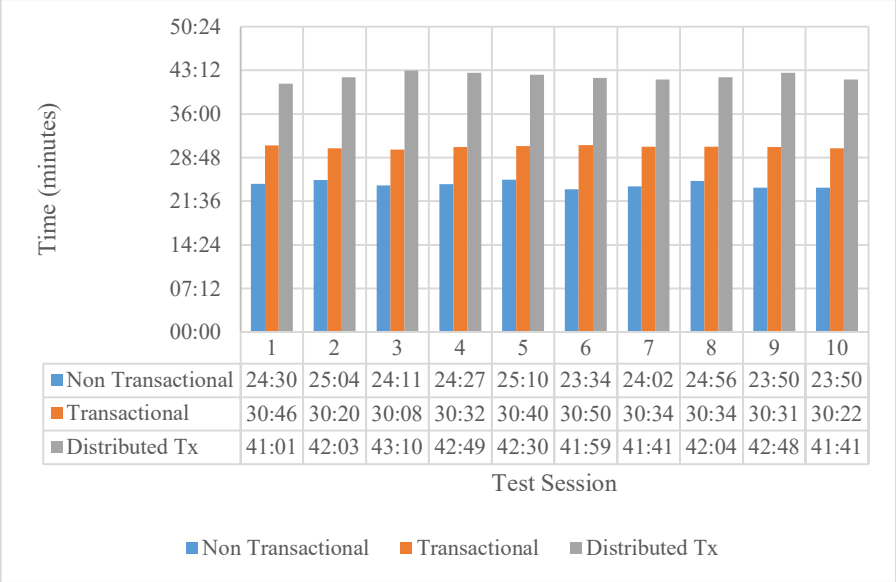

Figure 16: Completion Times for Network Latency Tests

Table 12: Completion Time Summary for Network Latency Tests

Transaction	Minimum	Maximum	Average	Median
None	00:23:34	00:25:10	00:24:21	00:24:19
Transactional	00:30:08	00:30:50	00:30:32	00:30:33
Distributed Transaction	00:41:01	00:43:10	00:42:11	00:42:04

Table 13: Duplicate and Lost Record Summary for Network Latency Tests

Transaction	Node-A Delta	Node-B Delta	Total Delta
None	0	0	0
Transactional	0	0	0
Distributed Transaction	0	0	0

Appendix C – Experiment Hardware Configuration

i) Software

- Oracle VirtualBox version 5.0.20
- SQL Server 2012
- Windows 10 Home Edition
- Windows Server 2012 R2 Standard Edition

ii) Host-System Configuration

- Windows 10 Home Edition
- DDR3L 32.0 GB RAM
- Intel Core i7-4700MQ CPU 2,40GHz, 4 cores with 8 logical processors
- Killer E2200 Gigabit Ethernet Controller
- 750GB HDD 7200rpm

iii) Virtual-Machine Configuration

- Windows Server 2012 R2 Standard Edition
- DDR3L 8.0 GB RAM
- Intel Core i7-4700MQ CPU 2,40GHz, 2 logical processors
- Killer E2200 Gigabit Ethernet Controller
- 750GB HDD 7200rpm