Integrating Distributed Tracing into the Narayana Transaction Manager

Miloslav Žežulka[1], Ondřej Chaloupka[2] and Bruno Rossi[1][a]
[1]*Masaryk University, Faculty of Informatics, Brno, Czech Republic*
[2]*Red Hat, Brno, Czech Republic*

Keywords: System Transactions, Distributed Transactions, OpenTracing, Distributed Tracing.

Abstract: ACID transactions have proven to be a very useful mechanism of ensuring reliability of applications. Guaranteeing transactional properties effectively and correctly is a challenging task by itself. Furthermore, investigating transaction issues in a distributed environment is at least equally complex and requires systematic data collection and analysis. In this paper, we present mechanisms and concepts of distributed tracing with focus on the OpenTracing API and showcase our integration of tracing capabilities into the Narayana transaction manager. We show that the performance impact of tracing does not drastically decrease user application performance while providing useful information for the analysis of running transactions.

1 INTRODUCTION

Transaction processing is designed to maintain a system's integrity in a known, consistent state, by ensuring that interdependent operations on the system are either all completed or all cancelled (Bernstein, 2009). This simple, yet very powerful concept is still used in many present-day applications. Therefore, understanding the details of transaction processing gives us big benefits during troubleshooting in a great variety of systems, allowing to improve the overall reliability and resilience (Rossi et al., 2010; Kanti-Singha Roy and Rossi, 2014).

Research on system transaction processing is still active even after more than forty years of its introduction to the field of computer science. When transaction processing exceeds boundaries of one isolated system and is performed in an interconnected distributed environment, the study of such systems becomes even more demanding and time-consuming (Štefanko et al., 2019; Štefanko, 2018).

Collecting information systematically from a distributed transaction processing system would aid in reasoning about transaction from a global perspective, e.g. what paths transaction processing took, which nodes caused the failure of a transaction or what was the exact cause of the failure. The Narayana (Little et al., 2020) transaction manager, written in Java and used in projects such as the WildFly Java EE application server, which lacks exactly such instrumentation.

The main goal of this article is to achieve the integration of OpenTracing into the Narayana Transaction Manager (TM). Tracing can be fundamental in distributed patterns such as the Saga Pattern (Štefanko et al., 2019; Garcia-Molina and Salem, 1987a). There have been many attempts to collect information about transactions, some of which date more than fifteen years (S. Smith, 2005; M. Goulet, 2011). The concept itself is therefore not new and for example, Narayana itself contains a great number of logging statements. However, at least to our knowledge, this paper presents a new way of systematic information collection from a TM, more specifically ArjunaCore, the transactional engine of Narayana.

The paper is structured as follows: in Section 2 we review the background concepts of distributed transactions. In Section 3 we delve into distributed tracing, presenting OpenTracing. In Section 4 we present the integration of tracing integration into Narayana, together with performance testing. In Section 5 we provide the conclusions.

2 DISTRIBUTED TRANSACTIONS

A *transaction* represents a single unit of work treated coherently and reliably independent of other transactions. Transactions usually represent a change in

[a] https://orcid.org/0000-0002-8659-1520

Žežulka, M., Chaloupka, O. and Rossi, B.
Integrating Distributed Tracing into the Narayana Transaction Manager.
DOI: 10.5220/0010448200550062
In *Proceedings of the 6th International Conference on Complexity, Future Information Systems and Risk (COMPLEXIS 2021)*, pages 55-62
ISBN: 978-989-758-505-0

a shared resource. To ensure all the desired properties of transactions, we need a supervisor which will make sure all transactions are executed accordingly to the defined set of rules. We call such supervisor a *transaction manager (TM)*. When a TM determines that a transaction can complete without any failure we say that it *commits* the transaction. This means that changes to shared resources take permanent effect and are available to other parties. Otherwise, the TM is required to perform a *rollback* of the transaction which discards any possible changes already made by the transaction.

In the 1983 paper *Principles of Transaction-Oriented Database Recovery* (Haerder and Reuter, 1983), Andreas Reuter and Theo Härder coined the acronym *ACID* as shorthand for *Atomicity, Consistency, Isolation and Durability*. It turns out to be quite difficult to scale traditional, ACID-compliant database systems. One of the possible ways of building a more scalable system is *horizontal scaling* which consists in distributing shared resources and their load over multiple nodes. The problem is that if a transaction accesses data that is split across multiple physical machines, guaranteeing the traditional ACID properties becomes increasingly complex; for example, the *atomicity* property of the ACID quadruple requires a distributed commit protocol to be run on top of the whole processing.

2.1 CAP Theorem

CAP theorem, also known as the Brewer's theorem, was proposed by Eric Brewer(Brewer, 2000). CAP is short for *Consistency, Availability* and *Partition-Tolerance*. The main contribution this theorem brings is that a distributed system data store can't satisfy all three properties at once. As a matter of fact, Seth Gilbert and Nancy Lynch published a proof of the Brewer's conjecture(Gilbert and Lynch, 2002) in 2002. Let us briefly introduce each one of the properties.

Consistency. There must exist a total order on all operations such that each operation behaves as if it were completed at a single point in time. This is equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time.

Availability. For a distributed system to be continuously available, every request received by a non-failing node in the system must result in a response.

Partition-tolerance. The network will be allowed to lose arbitrarily many messages sent from one node to another. When a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost.

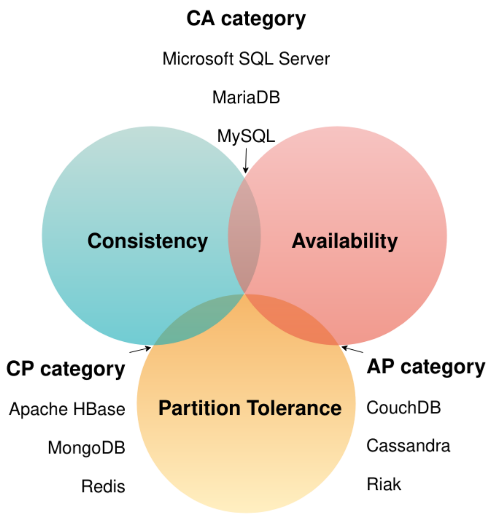

Figure 1: CAP theorem and all the possible compromise categories together with examples (as of 2015). Based on (Lourenço et al., 2015).

Since its inception, the CAP theorem has proven to be quite useful in helping distributed system designers to reason through capabilities of a proposed system. However, CAP has become increasingly misunderstood and misapplied. In particular, many designers incorrectly concluded that the theorem imposes certain restrictions on a distributed database system regardless of whether the system is operating normally or in a recovery mode. By strictly adhering to CAP, we design a system which is robust but might waste a lot of resources.

2.2 Sagas

In many scenarios, we need to process a series of transactions (possibly communicating among each other) which may as a whole last a long time, perhaps hours or even days. We call such transactions long-lived transactions (Garcia-Molina and Salem, 1987b; Štefanko et al., 2019) (LLT). Saga is an LLT transaction processing pattern and can be thought of as a metatransaction arching over multiple subtransactions[1].

To make a transaction atomic, a DTP system would usually lock all objects accessed by the transaction until it commits, thus preventing other transactions from accessing these resources and making them starve as a consequence. Such starvation is even more perceptible in LLTs. The solution sagas offer

[1]This does not imply that the meta-transaction adheres to all ACID principles, though. In sagas, each transaction is committed separately, meaning that sagas do not comply with the isolation property.

is to avoid locks on non-local resources, i.e. sagas hold only those resources needed by unfinished transactions. As soon as a subtransaction commits, it releases all of its local resources irrespective of whether other subtransactions have been committed or not.

We would still like to achieve atomicity in sagas, i.e. either all subtransactions are committed and the whole saga succeeds or at a certain stage, any subtransaction can fail and the whole saga needs to be aborted. Since some of the subtransactions may have already committed their work, we need a mechanism to *semantically* reverse the effect of any committed transaction. We call such a process *compensation*. Sagas use *compensating transactions* associated with every sub-transaction. Saga coordination then runs compensating transactions for all committed transactions in case of abort, thus accomplishing the atomicity property of ACID on the level of a saga.

Sagas utilize an alternative model which favours CAP availability over ACID isolation. Where ACID is pessimistic and forces CAP consistency at the end of every transaction, BASE is optimistic and accepts that the database consistency will be in a state of flux. Although this model has its indisputable disadvantages, BASE leads to levels of scalability that cannot be obtained with ACID. The availability of BASE is achieved through supporting partial inconsistencies without total system failure (Pritchett, 2008).

Having two or more participants as part of a transaction makes the transaction a distributed one. With resource-local transaction processing, a resource manager could process transaction updates independently of each other (Bernstein, 2009, p. 223). The global transaction must either commit updates of all resource managers or none. Such coordination issue belongs to a much more generic family of problems called *consensus*. One of the most known consensus protocols is the *two-phase commit* (2PC) protocol (Skeen, 1982).

3 DISTRIBUTED TRACING

So far, we have discussed how transaction state is transferred among all participants. We now also need a mechanism which would make it possible to transmit tracing data to various TMs since a Transaction Manager deals with XA transactions in a way that each node is managed by a standalone (but at the same time possibly subordinate in the context of a transaction) TM communicating with the rest of the distributed transaction processing environment.

We would like to have a mechanism supporting information propagation across multiple nodes. Moreover, we want to treat related information from various nodes as a whole. This can be extremely useful particularly for Narayana since we want to see how a particular execution of the 2PC consensus protocol behaves globally.

Distributed tracing shows a micro view of a request execution from the end-to-end perspective. As a consequence, we can retrospectively understand behaviour of each application component. Distributed tracing records everything related to an action in the system, i.e. captures detailed information of a request and all causally related activities (Stark et al., 2019). A collection of such activities is called a *trace*. For the purposes of this paper, one trace will be usually equivalent to one XA transaction.

X-Trace (Fonseca et al., 2007), one of the first frameworks which attempted to reconstruct a comprehensive view of service behavior, presented three main principles which were taken into consideration when designing it. These principles were later used in many relevant distributed tracing systems such as Dapper (Sigelman et al., 2010) or Canopy (Kaldor et al., 2017): i) The trace request should be sent in-band, rather than in a separate probe message, ii) the collected trace data should be sent out-of-band, decoupled from the original datapath, iii) the entity that requests tracing is decoupled from the entity that receives the trace reports.

Event causality in tracing is based on the Lamport's *happens before* relation (Lamport, 1978) and in such a way tracing *"[...] does not rely on clocks but on the sequence of event execution. Because of the propagation of metadata among related events, [X-Trace] captures true causality, rather than incidental causality due to execution ordering"* (Fonseca et al., 2010). Even though traces contain contextual data, one caveat with using tracing for troubleshooting is that it requires a global view yet granular attribution. For an enough large system, this might present an information overload since *"only a small fraction of data in each trace may be relevant to each engineer [...]"* but at the same time, *"traces [, by design,] contain all of the data necessary for anybody"* (Kaldor et al., 2017).

3.1 OpenTracing API

OpenTracing is a distributed tracing standard. In short, OpenTracing translates what previous tracing projects did, especially Google's Dapper (Sigelman et al., 2010), into a well-defined API[2].

[2]https://opentracing.io/specification/

OpenTracing is not the only option at hand [3]. First most distinguishable aspect around OpenTracing is a public list of OpenTracing-related projects called OpenTracing Registry (the Registry). The Registry is a collection of ”tracers, instrumentation libraries, interfaces, and other projects”. Besides this, the Registry also shows how active OpenTracing is, both in terms of its use in projects and development of the API itself. One other positive (not that much distinguishable from alternatives as the Registry, though) aspect of OpenTracing is that it supports the vast majority of mainstream languages. At the time of writing this paper, nine programming languages were officially supported: C#, C++, Go, Java, JavaScript, Objective-C, PHP, Python and Ruby. The project ”provides a vendor-neutral specification and polyglot APIs for describing distributed actions. [...] From an API perspective, there are three key concepts: `Tracer`, `Span`, and `SpanContext`”.

`Trace` is a directed graph the vertex set is a set of reported spans and the edge set is a set of references among respective spans. The OpenTracing standard does in no way prescribe any further constraints the structure of a trace. A trace is not directly represented in API and is represented as a set of `Span` instances.

There are two kinds of references: `ChildOf` and `FollowsFrom`, both representing a child-parent relation. In a `ChildOf` reference, the parent `Span` depends on the child `Span` in some capacity. For example, a `Span` representing the server side of an RPC may be the `ChildOf` a `Span` representing the client side of that RPC. Some parent `Spans` do not depend in any way on the result of their child Spans, e.g. when we model asynchronous behaviour like emitting and receiving messages to the message bus. The OpenTracing standard describes such causality as a *follows from* relation. `SpanContext` depicts a state of a `Span` that needs to be propagated to descendant `Spans` and across process boundaries. Having such a representation in the API is basically what makes the OpenTracing standard *distributed*. The idea of distributed context propagation resides in associating certain metadata to every request that enters the system, then propagating this metadata across thread, process or machine boundaries as the request gets fanned out to other services (Popa and Oprescu, 2019). The `Tracer` interface creates `Spans` and understands how to *inject* (serialize) and *extract* (deserialize) their metadata across process boundaries. It has the following capabilities: start a new `Span`, inject a `SpanContext` into and extract a `SpanContext` from a carrier which acts as an abstraction over any environment through which data is sent. A trace can be represented and visualized in multiple ways. In Fig. 2, we can see two different representations of the same trace. The first, causal representation, depicts the relationship among spans. The latter temporal visualisation focuses more on the time domain at which the trace spans were recorded, making it more useful when debugging latency-related issues in the application.

[3]See OpenZipkin - https://zipkin.io, OpenCensus - https://opencensus.io

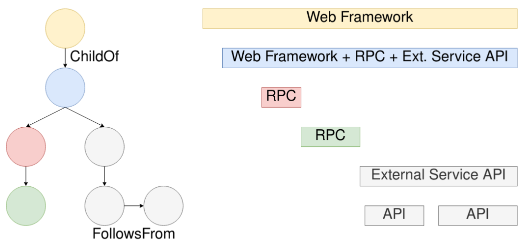

Figure 2: Causal and temporal relationships among spans of a same trace. Adapted from: https://opentracing.io/docs/best-practices/instrumenting-your-application/.

4 TRACING INTEGRATION IN NARAYANA

We delve now into the main contribution of the paper, that is the integration of OpenTracing into Narayana.

4.1 Narayana Transaction Processing

Narayana (https://narayana.io) is a library which provides an implementation of Java Transaction API (JTA). In a broader context, though, we can think of Narayana as a transaction toolkit providing a broad range of standards-based transaction protocols and mechanisms, namely JTS, WS-AT, WS-BA, REST-AT Transactions, STM, XATMI/TX and LRA.

4.1.1 Architecture Overview

The centrepiece and core part of Narayana is called ArjunaCore. There are three main ArjunaCore components. The first part is the TM engine itself and can be thought of as a state machine. The second part is a recovery manager running as a separate process. The last unit is responsible for persisting auxiliary information related to transaction processing, also known in the context of Narayana as the *object store* (or object storage).

ArjunaCore is used as a transaction engine for the above-mentioned protocols which are system-wise one layer above ArjunaCore. For the purposes of this paper, we will focus purely on ArjunaCore.

`StateManager` provides primitive facilities necessary for managing persistent and recoverable objects. These facilities include support for the activation and deactivation of objects, and state-based object recovery. `LockManager` represents the concurrency controller of the engine. `Lock`, a synchronization primitive specific to ArjunaCore, is made persistent. Locks are maintained within the memory of the virtual machine which created them.

As the name suggests, `AbstractRecord` is an overarching abstract class which defines an interface used in ArjunaCore to notify objects that various state transitions have occurred as an XA transaction executes. In other words, any class having a subtype `AbstractRecord` acts as an intermediary between a transaction processing and a related record in the object store. ArjunaCore remembers object state: this serves two purposes. Firstly, the recovery manager can then use such information for transaction recovery. The second, directly related one, is persistence. The state represents the final state of an object at application termination.

The *failure recovery subsystem* of Narayana will ensure that results of a transaction are applied consistently to all resources affected by the transaction, even if any of the application processes or the machine hosting them crash or lose network connectivity. In the case of machine (system) crash or network failure, the recovery will not take place until the system or network are restored, but the original application does not need to be restarted – recovery responsibility is delegated to a recovery manager which runs as a separate process(Little et al., 2020). The main responsibility of the recovery manager is to periodically scan the object store for transactions that may have failed; failed transactions are indicated by presence in an object store after a timeout that the transaction would have normally been expected to finish. Narayana also utilizes a so-called *reaper thread* which separately and periodically monitors all locally created transactions and forces them to roll back if their timeouts elapse.

4.2 Integration Design

One of the most crucial stages of any tracing integration is to properly understand its domain. Without such preparation, we render tracing useless.

The first important task, therefore, was to design a generic trace structure reflecting the way Arjuna processes a GT. In our integration, each GT is represented by an OpenTracing trace and vice versa. Structure of such a trace can be seen in Fig. 3. To construct a complete trace, we needed to find appropriate code insertion points inside Narayana which would correspond to an action done by the TM. Narayana contains a great deal of logging statements and after careful inspection of logs, we were able to pinpoint most of the code locations. In Table 1, we can see final choices of instrumentation points in ArjunaCore.

Table 1: Span delimiting events in ArjunaCore.

Span action	Instrumentation point
global prepare	`BasicAction.prepare`
branch prepare	`XAResourceRecord.topLevelPrepare`
global commit	`BasicAction.phase2Commit`
branch commit	`XAResourceRecord.topLevelCommit`
global abort	`BasicAction.phase2Abort`
branch abort	`BasicAction.doAbort`
enlistment	`TransactionImple.enlistResource`

4.3 Problems Encountered

During tracing integration itself, we encountered many blind alleys and needed to rethink some of the design choices. Let us now talk about the most important items we needed to deal with.

4.3.1 Cross-thread Manual Context Transfer

During the first draft of the implementation, we used a VM-global `ConcurrentHashMap` which we used for transfer of `Spans` across multiple threads, where the map key was a `String` representation of a GT identifier. This way, every event in Narayana has the ability to be attached to this span and together, we get a complete trace irrespective of on which threads the processing happened.

We can already see that this approach is a bit ineffective as every access to this global storage might be synchronized. In the final integration, this has been mitigated by associating every root span (as a private class attribute) to a `BasicAction` which is where transaction processing both starts and ends. This way, the whole lifecycle of a span can be managed from one thread. Nevertheless, even if we, for some reason, needed to transfer spans across thread boundaries, this would be still best done by `SpanContext` injection and extraction among threads. Choosing global collection is bound to be very ineffective and prone to errors related to ”memory leaks” (e.g. spans which will never be deleted from the global collection because of a transaction processing path which the integration has not dealt with or even has not expected).

4.3.2 Fluid OpenTracing API

OpenTracing API for Java is three years old but even during the work on the integration, substantial

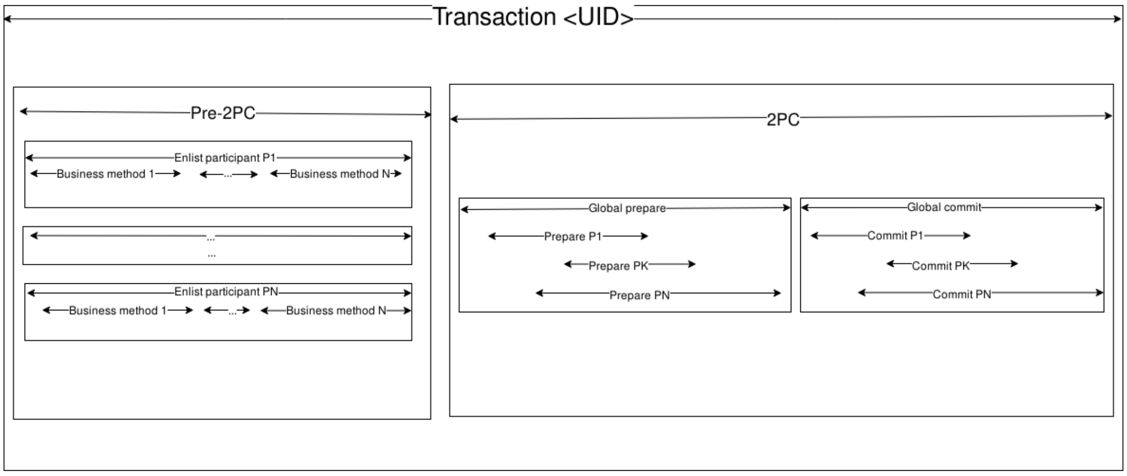

Figure 3: Causal and temporal relationships among spans of a same trace. Adapted from: https://opentracing.io/docs/best-practices/instrumenting-your-application/.

changes were still being made, e.g. how a tracer is registered or whether spans were subject to automatic resource management, in Java realized as a `try-with-resources` statement.

Frequent changes in the API were also reflected in obsolete tutorials at the OpenTracing project website, as of the beginning of 2020, making it more difficult for a new user to use properly. We created a specialized `narayanatracing` module in Narayana which tries to avoid such complications. This means that all the necessary changes forced by the change of the API will be only propagated at only one place. The other motivation behind introducing the module is to ease the use of tracing as much as possible. We know in advance what the structure of a transaction span will be and we can use this to our advantage when using tracing inside Narayana.

4.4 Performance Testing

In a time-critical component which transaction manager certainly is, we need to measure the overhead caused by introducing the tracing code. If the overhead is too large, there is no practical use to such instrumentation and it might be useful to resort to already existing practices instead.

When measuring tracing overheads, engineers from Google in their Dapper paper argued that *"the cost of a tracing system is felt as performance degradation in the system being monitored due to both trace generation and collection overheads, and as the amount of resources needed to store and analyze trace data"* (Sigelman et al., 2010). Tracing was from the very beginning designed in a way that *"... propagating metadata on its own is unlikely to become a bottleneck. Generating reports, however, could become a significant source of load"* (Sigelman et al., 2010).

At first sight, it might seem that the only overhead caused by the tracing is the time it takes to handle all the OpenTracing API calls and optionally, how long does it take to propagate a span across the network.

In general, this seems to be the case but we also need to take into consideration a situation in which the tracing is present in the codebase but the tracing itself has not been activated. What Jaeger does by default is to register a *no operation* tracer (no-op tracer) which serves as a dummy implementation of the tracer which, as the name suggests, does not perform any real action and immediately returns from any of its API calls.

Even though the analogy of tracing with logging fails on many levels, it is still useful to compare the differences of overhead between those two approaches. Narayana contains a great deal of log statements for debugging and development purposes. Therefore, we use a special logger, the only one turned on during performance testing, which bypasses the standard Narayana logging; this way, we have direct control over what is logged. All the logging statements are written to a single log file persisted to a local file system. We tested five types of Narayana:

- T1. `file-logged`: Narayana with logging statements transformed from the tracing instrumentation as described above),
- T2. `jaeger`: with tracing code and an activated Jaeger tracer,
- T3. `noop`: with tracing code inside but with only a default *no operation* tracer registered,
- T4. `tracing-off`: tracing code is present but its execution is completely circumvented using if statements,

- T5. `vanilla`: cloned from the official repository with no changes in the code.

There are four main use cases tested in benchmarks: commit 2 XA resources, commit an XA resource, user-initiated rollback and a scenario in which an XA resource fails to prepare. It is difficult to reproduce a production environment with multi-threading concurrent access and all the impact it can have on performance. In order to address these issues, the OpenJDK project has developed a tool dedicated to benchmarks for languages running on the JVM called Java Microbenchmark Harness (JMH). This framework aims at executing automatically repeated executions of any chosen code and collecting statistics about code performance. The most basic usage of JMH is to write a class or classes containing a series of methods annotated with the `@Benchmark` annotation and JMH takes care of the rest of the setup. The resulting artifact is a standalone JAR file which is, very simply put, a JHM wrapper around the attached benchmark tests.

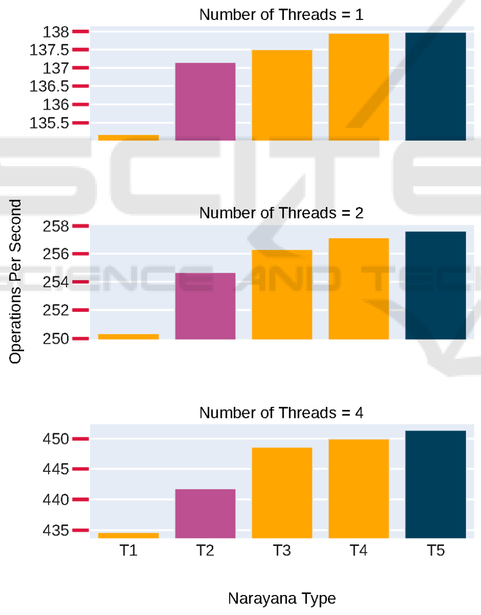

Figure 4: Performance test results (note different scales for better visual inspection).

4.4.1 Performance Troubleshooting

Once we conducted the first benchmark's runs we encountered an inexplicable and drastic decrease in performance; benchmark running Narayana patched with tracing performed roughly five to six times slower than upstream version of Narayana with no changes. We investigated this issue by running a profiler and examining whether there were any bottlenecks outside of the instrumentation itself. Indeed, the bottleneck was not caused by tracing but by using costly `toString` method on `XAResourceRecord` instances which were used quite often as a value for `XID` span attributes. Apart from this, we have also seen big overhead used by a backing collection inside Jaeger which held unreported spans; in the latest performance test, this overhead was reduced.

After running tests multiple times with the same stabilized configuration, we observed that performance remained practically invariant to the type of test we were running; the only factor which changed in respect to operations ArjunaCore was executing were *absolute* values, not ratios among various Narayana types. To illustrate our results, we've chosen a scenario in which two dummy XA resources are enlisted into the transaction and the transaction commits successfully. As we can see from Fig. 4, the overhead caused in T2 is in terms of tenths to units of percents[4]. However, we saw an increasing overhead as the number of threads went up. This was most probably be caused by necessary synchronization needed for accessing the backing collection, common to all threads, to which spans are stored until finished and reported. Therefore, most of the overhead is not caused by execution of tracing code itself.

5 CONCLUSION

The main goal of this article was to show the integration of distributed transaction tracing into a transaction manager (the Narayana platform). We measured the impact of tracing in terms of several alternative implementations. Moreover, we provided a proof of concept which enables us to connect information of transaction processing spanning over multiple nodes, thus showing the full potential of distributed tracing in terms of OpenTracing.

We integrated the OpenTracing API into the Narayana transaction manager in a way that it provides enough information to discover and monitor transaction problems. Overall, the results show that the integration does not hinder performance in any noticeable way. During the journey to the state-of-the-art integration, we were able to investigate main concepts behind OpenTracing and use such knowledge for its integration into the Narayana transaction manager. We also listed various obstacles which can stand in a way of any software developer attempting to integrate tracing into other projects.

[4]The exact overheads ratios are (starting from one thread counting upwards) are 0.6%, 1.2% and 2.2% using the formula $100 \times (1 - \frac{T2}{T5})$.

ACKNOWLEDGMENT

The research was supported from ERDF/ESF "CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence" (No. CZ.02.1.01/0.0/0.0/16_019/0000822).

REFERENCES

Bernstein, P. (2009). *Principles of transaction processing*. Morgan Kaufmann Publishers, Burlington, Mass.

Brewer, E. A. (2000). Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 7, New York, NY, USA. Association for Computing Machinery.

Fonseca, R., Freedman, M. J., and Porter, G. (2010). Experiences with tracing causality in networked services. *INM/WREN*, 10(10).

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. (2007). X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Univ. of California, Berkeley.

Garcia-Molina, H. and Salem, K. (1987a). Sagas. *ACM SIGMOD Record*, 16(3):249–259.

Garcia-Molina, H. and Salem, K. (1987b). Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, page 249–259, New York, NY, USA. Association for Computing Machinery.

Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, page 51–59.

Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, page 287–317.

Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., and Viscomi, B. (2017). Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 34–50, New York, NY, USA. Association for Computing Machinery.

Kanti-Singha Roy, N. and Rossi, B. (2014). Towards an improvement of bug severity classification. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 269–276.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Little, M., Halliday, J., Dinn, A., Connor, K., Musgrove, M., Robinson, P., Trikleris, G., and Feng, A. (2020). Narayana project documentation.

Lourenço, J., Cabral, B., Carreiro, P., Vieira, M., and Bernardino, J. (2015). Choosing the right nosql database for the job: a quality attribute evaluation. *Journal of Big Data*, 2:18.

M. Goulet, S. T. Rader, A. S. (2011). Selective reporting of upstream transaction trace data. US Patent 8,392,556.

Popa, N. M. and Oprescu, A. (2019). A data-centric approach to distributed tracing. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 209–216.

Pritchett, D. (2008). Base: An acid alternative. *Queue*, 6(3):48–55.

Rossi, B., Russo, B., and Succi, G. (2010). Modelling failures occurrences of open source software with reliability growth. In Ågerfalk, P., Boldyreff, C., González-Barahona, J. M., Madey, G. R., and Noll, J., editors, *Open Source Software: New Horizons*, pages 268–280, Berlin, Heidelberg. Springer Berlin Heidelberg.

S. Smith, D. Schank, M. T. (2005). System and methods for cross-tier transaction tracing. US Patent 7,886,281.

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. *Google Technical Report*.

Skeen, D. (1982). A quorum-based commit protocol. Technical report, Cornell University, USA.

Stark, S., Sabot-Durand, A., Loffay, P., Mesnil, J., Rupp, H. W., and Saavedra, C. (2019). *Hands-On Enterprise Java Microservices with Eclipse MicroProfile*. Packt Publishing Ltd., S.l.

Štefanko, M. (2018). Use of transaction within a reactive microservices environment. Master's thesis, Masaryk University, Brno.

Štefanko, M., Chaloupka, O., and Rossi, B. (2019). The saga pattern in a reactive microservices environment. In *Proceedings of the 14th International Conference on Software Technologies - Volume 1: ICSOFT,*, pages 483–490. INSTICC, SciTePress.