Hyflow2: A High Performance Distributed Transactional Memory Framework in Scala

Alexandru Turcu
Virginia Tech
talex@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Abstract

Distributed Transactional Memory (DTM) is a recent but promising model for programming distributed systems. It aims to present programmers with a simple to use distributed concurrency control abstraction (transactions), while maintaining performance and scalability similar to distributed fine-grained locks. Any complications usually associated with such locks (e.g., distributed deadlocks) are avoided. We propose a new DTM framework for the Java Virtual Machine named Hyflow2. We implement Hyflow2 in Scala and base it on the existing ScalaSTM API soon to be included in the Scala standard library. We thus aim to create a smooth transition from multiprocessor STM programs to DTM.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming–distributed programming; D.1.3 [*Programming Techniques*]: Concurrent Programming–parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features–concurrent programming structures

General Terms Languages, Performance.

Keywords transactional memory, distributed systems, nested transactions, open nesting

1. Introduction

Programming distributed concurrency has always been a difficult task. Today, there are three popular models that can be used to address such a task: shared memory, actors and transactions.

In the **shared memory model**, processes access the memory representing the shared state while ensuring safety using synchronization primitives such as distributed locks. This model is supported by technologies such as RPC and RMI that allow remotely invoking methods on objects (this is know as the *control-flow* model, because the computation moves where the data is). Synchronization primitives are available using dedicated platforms like Apache Zookeeper [7] and Hazelcast [3] or can be implemented ad-hoc. Alternatively, in the *data-flow* model, distributed caches such as Ehcache [2] and Infinispan [4] can be used to bring the data where the computation is. The shared memory model however is prone to hard-to-trace concurrency bugs such as race conditions, dead-locks and live-locks.

The actor model prohibits sharing memory by encapsulating mutable state inside light-weight sequential constructs called actors. Actors communicate via message passing and their operations always execute sequentially, thus avoiding concurrency problems. The actor model is based on Communicating Sequential Processes (CSP) introduced by Hoare in [13], and became popular with the advent of the Erlang programming language. Since then, many languages (e.g. Scala and Google Go) and frameworks (e.g. Akka, ActorKit) have embraced this model. The actor model is very effective when applicable, but some problems are difficult to formulate within its restrictions. Furthermore, it requires changing the way most programmers think about concurrency.

Transactions are the preferred concurrency mechanism in database environments. They provide ACID properties (Atomicity, Consistency, Isolation and Durability), making them easier to reason about compared to low-level primitives (locks) or even actors. Transactions are sequences of operations that either all execute successfully or all fail. A failed (aborted) transaction has no effects visible to other transactions (its operations are *rolled-back*). A successful (committed) transaction appears to take effect atomically, and any changes performed while the transaction is running are not visible to other committed transactions.

On the downside, distributed transactions do not seamlessly integrate with popular programming languages. The most common approach is to delegate all transactional processing to a separate database server. A client library would then be used to communicate with the database server, sending it commands expressed in the Structured Query Language (SQL) and receiving the result of their execution.

Writing SQL can be avoided by employing an additional software layer called an Object Relational Mapper (ORM), further increasing complexity.

Programmers wanting to use transactions for their distributed applications can also employ the *X/Open XA* standard or the equivalent Java Transaction API (JTA). While designed to coordinate multiple transactional resources (such as database servers or message queues) in distributed transactions, XA/JTA can be used to provide distributed transactional access to regular, in-memory objects. Alternatively, recent distributed cache frameworks provide transactional access to their stored data.

Significant effort has been spent in the multiprocessor research community towards Transactional Memory (TM). TM is an abstraction that aims to replace locks as a synchronization primitive with transactions. Many TM systems use *atomic blocks* to enclose code that must execute atomically. In-memory transactions are utilized behind the scenes, but in many cases, the user does not need to be aware of it. Aborted transactions are implicitly retried until they succeed.

We believe distributed concurrency should be seamlessly expressed in a programming language just like atomic blocks succeed to do for multiprocessor concurrency. Furthermore, many applications do not need durability (or have relaxed requirements for durability) so employing a classic disk-backed relational database is an overkill. Distributed Transactional Memory (DTM) addresses these issues. The DTM model was proposed [12] to replace shared memory systems using distributed locks with light-weight, in-memory transactions.

This paper describes our new DTM implementation for the Java Virtual Machine (JVM) named Hyflow2. Hyflow2 is available as an open-source project at our website, *http://hyflow.org/*. Hyflow2 is a complete rewrite of our previous DTM library, Hyflow [16, 17], which is also available at the same website. In designing Hyflow2 we focused on several issues that could be improved compared to the original Hyflow: modularity, clean API that does not require byte-code rewriting, and performance.

Hyflow2 is written in the Scala programming language for the JVM and internally uses the actor concurrency model by employing the Akka [1] toolkit. We provide two APIs: a Scala API that uses Scala's powerful control abstractions and a Java API for compatibility. The Scala API is based on the excellent ScalaSTM API [6, 9], which is due to be included in Scala's standard library. Hyflow2 is a library DTM: it requires no compiler or run-time support. This enables easy deployment on standard JVMs.

Hyflow2 currently provides an implementation of the Transactional Forwarding Algorithm (TFA, [18]), a DTM technique that uses the data-flow model (immobile transactions, mobile objects). We support the flat, closed and open nesting models [20, 21], as well as distributed conditional synchronization.

```
@Atomic
void transfer(Account a1, Account a2, int amount)
{       withdraw(a1, amount);
        deposit(a2, amount);
}
@Atomic
void withdraw(Account a, int amount) {
        a.value -= amount;
}
@Atomic
void deposit(Account a, int amount) {
        a.value += amount;
}
```

Figure 1. Example of the original Hyflow API. Transactions are marked using the @Atomic annotation.

To the best of our knowledge, Hyow2 is the first Distributed Transactional Memory implementation with support for Scala, interoperability with Java, and key DTM features including nested transactions and distributed conditional synchronization. Our focus on performance lead to significant speed improvement compared to Hyflow. In our tests, Hyflow2 proved up to 7 times faster at low node counts and up to 100% faster at high node counts.

The remainder of the paper is organized as follows. Section 2 briefly describes the original Hyflow library and the areas where it was lacking. Section 3 introduces Hyflow2's new API. Section 4 describes transactional nesting and the API for supporting it in Hyflow2. Implementation is discussed in Section 6. Hyflow2 is experimentally evaluated in Section 7. Related work is briefly mentioned in Section 8 and Section 9 concludes the paper.

2. The Hyflow DTM framework

Hyflow is our original DTM research prototype. It was built on top of the Deuce STM library and the Aleph communication framework, two research projects that are not actively maintained. Hyflow's modular design attempts to allow for pluggable network transports, transactional algorithms, directory protocols and contention managers. The interfaces to the various components were however not flexible enough, and hard-coded links were introduced between some of the modules' implementations. Additionally, we were compelled to make some changes to the two libraries we're using in order to work around their limitations. Eventually, the source code became difficult to maintain and in dire need of restructuring, thus leading us to start working on Hyflow2.

Hyflow (just like the underlying Deuce STM) relies on automatic byte-code rewriting to provide an API based on annotations. As seen in Figure 1, the user marks the methods to be executed transactionally as *@Atomic*. A Java Agent rewrites such methods into two polymorphic copies: the

first copy has the same signature as the original method, and it initiates a new transaction (or reuses an already running transaction, if available) and then calls the second copy within the context of this transaction. The second copy is a transacted version of the original method's byte-code. It takes an additional argument (a transaction context), and replaces all field reads and writes with transactional read and write operations. Any method calls within transacted code are modified to also pass the transaction context argument.

The automatic instrumentation also touches on methods not marked as *@Atomic*, by creating an additional transacted copy of the method as described above. When such method is called outside any transaction, the original byte-code is executed. When methods are called within a transaction (by transacted code), the addition of the transaction context argument leads to executing the transacted versions of the methods.

This approach works particularly well for a simple multiprocessor transactional memory system because the instrumented byte-code can be made very fast: no extra objects need to be instantiated (the transactional context object can be reused), method calls can be kept to a minimum (the transactional read and write operations can be inlined), and only one thread-local variable lookup needs to be performed at the beginning of the transaction. However it is a nightmare to develop new features under this model: the instrumented byte-code cannot readily be debugged, while working on the Java agent performing the instrumentation is the equivalent of programming in assembly! Moreover, the potential speed benefits of this model become negligible when dealing with distributed systems, where network accesses are the most costly operations. Modern JVMs with state-of-the-art Just-in-Time (JIT) compilation and garbage collection further minimize the benefits of the byte-code rewriting approach.

Instead, our effort is better spent optimizing the real bottlenecks of the system: network round-trip time and thread context switch overheads. We will further explain these issues in Section 6.6.

3. Hyflow2 API

Hyflow2 API is based on the excellent ScalaSTM API[6]. In fact, Hyflow2 tries to reuse ScalaSTM's interfaces wherever possible, and partially implements a back-end for the ScalaSTM API.

3.1 ScalaSTM

ScalaSTM is an STM API for Scala due to be included in the Scala standard library in an upcoming release. The API allows for pluggable back-end implementations, and it ships with a reference implementation, CCSTM[9]. Hyflow2 inherits all features described in this section.

Transactions in ScalaSTM are defined using *atomic blocks*, as shown in Figure 2. To achieve this syntax, *atomic*

```
val ctr = Ref(0)
atomic { implicit txn =>
  ctr() = ctr() + 1
}
```

Figure 2. An example transaction in ScalaSTM (common usage).

```
val ctr: Ref[Int] = Ref[Int](0)
atomic.apply(new Function1[InTxn,Unit] {
  def apply(implicit txn: InTxn): Unit = {
    ctr.update(ctr.apply(txn) + 1)(txn)
  }
})
```

Figure 3. A more verbose version of the code in Figure 2, with several Scala syntactic shortcuts written explicitly.

is a *TxnExecutor* object whose *apply* method takes a function as its only argument and executes this function as a transaction. The "implicit txn =>" construct denotes that the function passed to *apply* takes one implicit argument, the transaction context object.

ScalaSTM uses *transactional references* (*Ref*s) as a container for the values that are to be accessed using transactional semantics. The Ref containers mediate all access to the data within. To access a value of a Ref *ref1* within a transaction, one would use *ref1()* – i.e., call ref1.apply() – or ref1.get() as an alternative syntax. To change the value of the Ref inside a transaction, one should use *ref1() = v* – i.e., call ref1.update(v) – or alternatively, *ref1.set(v)*.

All of these methods (apply, get, update and set in class Ref) take a transaction context object (i.e., an instance of the class *InTxn*) as an additional, implicit argument. Implicit arguments in Scala code may be omitted, as long as the compiler can find in scope a variable of the appropriate type marked with the *implicit* keyword. In Figure 2, the *txn* object is automatically passed to the apply() and update() methods. Figure 3 shows how Scala interprets the code in Figure 2.

This mechanism using implicit arguments and Refs leads to a clean syntax with relatively little redundant code (only the "implicit txn =>" construct and the function call "()" characters are superfluous). Another benefit of this mechanism is *strong atomicity* for all Refs. Strong atomicity is the desirable property of a TM system which protects against concurrent access of a memory location from both transactional code and non-transactional code (for contrast, a *weakly atomic* TM system would have an undefined behavior in this situation). Accesses to a Ref's contents via the *apply* or *update* methods require an implicit transaction context object to be in scope, otherwise compilation fails. This requirement is satisfied inside an atomic block as explained in the previous paragraph. Outside atomic blocks

```
def takeFirst(): T = atomic {
  implicit txn =>
    val old_head = this.head()
    if (old_head == null)
      retry // do not proceed if empty
    this.head() = old_head.next
    return old_head.value
}
```

Figure 4. Conditional synchronization using retry. Transaction can only proceed once there is at least one item in the list.

```
class Account(val _id: String) extends HObj {
  val type = field("") // a string field
  val value = field(0) // an integer field
  Hyflow.dir.register(this) // Register with the
      directory manager
}
```

Figure 5. Hyflow2 Object example for a bank account.

however, no transaction context value is implicitly available, so calls to *apply* or *update* would lead to compilation errors. Single-operation transactions are used to allow accessing Refs outside atomic blocks. *ref1.single.get()* would, for example, spawn a transaction for the sole purpose of retrieving ref1's value.

ScalaSTM allows temporarily aborting a transaction using the *retry()* method. This is usually used for enforcing preconditions. Suppose for example the *takeFirst* operation on a queue (Figure 4). When the queue is empty, this operation may invoke retry, effectively blocking until at least one element is available. This behavior is called *conditional synchronization*. After calling *retry*, the transaction should only execute again once any of the values it has read is updated, otherwise it will follow the same execution path and call *retry* again. A simplistic implementation may, however, blindly restart the transaction after an exponential back-off.

3.2 Hyflow2 Objects

While in ScalaSTM transactions operate on Refs directly, Hyflow2 introduces an additional layer – the Hyflow2 Object – as a container for Refs (see Figure 5). An Hyflow2 Object mixes in the HObj Scala trait [1] and is Hyflow2's basic unit of data. Each Hyflow2 Object (henceforth referred to as HObj) has a unique identifier, which Hyflow2 uses to locate the object. The key is usually specified by the user at the object's creation, by passing it as an argument to the constructor.

Each HObj is composed from one or more fields. Fields are specialized Refs that maintain their association with the enclosing HObj and their order number within that object. Fields are created by calling the *HObj.field* method inside the object's constructor, and passing it an initial value.

[1] A Scala trait is similar to a Java interface. A class can therefore mix in (i.e., implement) multiple traits. However unlike interfaces, Scala traits may contain implementation.

```
def deposit(accId: String, amount: Int) = atomic {
  implicit txn =>
    val acc = Hyflow.dir.open[Account](accId)
    val newVal = acc.value() + amount
    acc.value() = newVal
    return newVal
}
```

Figure 6. Hyflow2 transaction example. Transaction must open an object before operating on it.

3.3 Hyflow2 Directory Manager

The Directory Manager (DM) is Hyflow2's module that keeps track of the objects' location. When an HObj instance is created, it registers itself with the DM (Figure 5). If the object later migrates to a different node, it updates its registration with the DM.

The Directory Manager also handles retrieving objects from their owner nodes over the network. This operation is called *opening* (see Figure 6). It requires the identifier of the requested object and it generally caches a copy of the requested object on the local node.

4. Transaction Nesting

Hyflow2 includes support for nested atomic blocks. In this section we first briefly describe the three nesting models previously studied in TM [11, 14]: flat, closed and open. Next we introduce the API support for nesting in Hyflow2, and explain how it works. Lastly, we make the case for a third atomic construct.

4.1 Nesting Models

The three transaction nesting models differ based on whether the parent and children transactions can independently abort:

Flat nesting
: is the simplest type of nesting, and simply ignores the existence of transactions in inner code. All operations are executed in the context of the outermost enclosing transaction, leading to large monolithic transactions. Aborting the inner transaction causes the parent to abort as well (i.e., partial rollback is not possible), and in case of an abort, potentially a lot of work needs to be rerun.

Closed nesting
: In closed nesting, inner transactions can abort independently of their parent (i.e., partial rollback), thus reducing the work that needs to be retried. Changes are only made visible to outside transactions when the outermost transaction commits.

```
// Simple open-nested transaction without
    abstract locks or commit or abort handlers
atomic.open { implicit txn =>
  val ctr = Hyflow.dir.open[Counter]("id")
  ctr.value() += 1
}
// Open-nested transaction that acquires a single
    abstract lock
atomic.open("abslock0") { implicit txn =>
  val ctr = Hyflow.dir.open[Counter]("id")
  ctr.value() += 1
}
// More complex usage case, with abort and commit
    handlers. Lock is held after commit.
atomic.open { implicit txn =>
  acquireAbsLock("absLock0")
  val ctr = Hyflow.dir.open[Counter]("id")
  ctr.value() += 1
} onAbort { implicit txn =>
  val ctr = Hyflow.dir.open[Counter]("id")
  ctr.value() -= 1
} onCommit { implicit txn =>
  holdAbsLock("absLock0")
}
```

Figure 7. Open nesting in Hyflow2

Open nesting

In open nesting, operations are considered at a higher level of abstraction. Open-nested transactions are allowed to make their changes visible and commit to the shared memory independently of their parent transactions, optimistically assuming that the parent will commit. If however the parent aborts, the open-nested transaction needs to run compensating actions to undo its effect. The compensating action does not simply revert the memory to its original state, but runs at the higher level of abstraction. For example, to compensate for adding a value to a set, the system would remove that value from the set. Although open-nested transactions breach the isolation property, this potentially enables significant increases in concurrency and performance. Open-nested transactions typically use constructs called *abstract locks* to guarantee consistency.

4.2 Nesting API

Flat and closed nesting are semantically equivalent and can be used interchangeably. Unlike in the original Hyflow, we decided not to expose the decision of which of the two models to use in the standard user-facing API. Hyflow2 may use any of these models to handle nested atomic blocks. Currently, the decision is fixed based on a configuration value, but in the future it could be made adaptively at runtime.

```
new OpenNestingBlock(
  atomic.open { implicit txn =>
    // Atomic bloc is wrapped in an
        OpenNestingBlock
  }
).onCommit( { implicit txn =>
    // handler is passed to onCommit method. After
        registering the callback, onCommit
        executes the block wrapped above.
  }
)
```

Figure 8. Expanded code showing mechanism for defining commit/abort handlers.

Open nesting on the other hand requires API support. Following the style of ScalaSTM, in Hyflow2 we propose the following syntax (see Figure 7):

- An open nested transaction should be started with *atomic.open* . The body of the transaction follows in braces, just like for regular transactions.
- Following the transaction's body two optional blocks may be specified. These blocks are introduced by *onCommit* and *onAbort*, and represent the transaction's commit and abort handlers, respectively. The handlers themselves are executed as open-nested transactions, so they must accept the implicit transaction context argument. If both handlers are present, their order is not important.
- If an open-nested transaction requires the acquisition of an single abstract lock which is known in advance, the lock's identifier can be passed as a string argument to *atomic.open* . The lock will be acquired before the open-nested transaction can commit, and will be released automatically as part of the transaction's abort and commit handlers. These handlers do not need to be present in the code, the lock will be released anyway (see Figure 7).
- For any other abstract lock scenarios, the locks must be acquired within the sub-transaction's body using *acquireAbsLock*. These locks too will be automatically released as part of the sub-transaction's abort and commit handlers.
- If for any reasons an abstract lock should be kept beyond the sub-transaction's commit or abort, *holdAbsLock* must be called in the commit and/or abort handler. Any such lock will be propagated to the innermost open-nested ancestor transaction and will be released upon its commit or abort.

4.3 Discussion and Language Mechanisms

We consider *atomic.open* a semantically cleaner way of denoting open-nesting transactions than the previously suggested *openatomic* keyword [15]. Our syntax logically breaks

down into two terms. The first term, *atomic* is the same as the marker for regular atomic blocks. The second term, *open*, appears as a property of the resulting transaction. By contrast, *openatomic* as a separate keyword, gives the impression the effect is totally unrelated with that of the *atomic* keyword.

When evaluating an *atomic.open* block, the *open* method is called on the *atomic* object of type *TxnExecutor*, and it receives the function to be executed transactionally as a parameter. Declaring the *onCommit* and *onAbort* handlers is more complex: blocks are evaluated last to first, wrapping what is above in a special *OpenNestingBlock* container object, and calling *onCommit*/*onAbort* on this object. The object is saved in a thread-local variable. When finally, *atomic.open* is invoked, it checks if there is any *OpenNesingBlock* object registered for the current thread and uses it, if any. See Figure 8 for an expanded example. This mechanism is also used in ScalaSTM to implement the *orElse* keyword (*orElse* provides the means to execute alternative atomic blocks if the original ones fail).

4.4 Configurable nesting

For testing reasons users may need to execute certain atomic blocks under both flat/closed and open nesting models. Using the previously described API, switching between models would require modifying the source code and recompiling. To avoid this situation, we support an additional method for launching a transaction, "*atomic.config*". An atomic block marked with *atomic.config* will determine its nesting model at run-time, by reading it from a configuration value. For completeness, the choice between flat and closed nesting is explicit. *Atomic.config* allows defining abort and commit handlers just like *atomic.open*. If the block executes with closed or flat nesting, these handlers will simply be ignored.

5. Java Compatibility API

Scala provides excellent interoperability with Java. As a result, many of the operations described above will just work when invoked from Java code either directly, or in a slightly different form (for example, methods *ref1.get*, *ref1.set*, *Hyflow.dir.open*, *retry* becomes *Txn.retry*, etc.). Several of the more advanced Scala features that we use in the Hyflow2 API are however not supported from Java code, so we need to provide additional mechanisms to obtain the same results.

5.1 Defining Transactions

ScalaSTM already provides a way for starting transactions from Java which uses the Callable and Runnable interfaces for defining the transaction's body (Figure 9). The transaction context argument isn't used anymore – instead, all transactional operations need to dynamically determine the context object at run-time. If no transaction exists for the current thread, a single-operation transaction is created automatically. This mechanism, however, does not define the abort and commit handlers required for open-nesting.

```
STM.atomic(new Runnable {
  public void run() {
    Counter ctr = Hyflow.dir().<Counter>open("ctr")
    ctr.set(ctr.get() + 1);
} } );
```

Figure 9. ScalaSTM Java compatibility API.

```
new Atomic<Boolean> {
  public Boolean atomically(InTxn txn) {
    Counter ctr =
        Hyflow.dir().<Counter>open("ctr");
    ctr.value.set(ctr.value.get() + 1);
    return true;
  }
  public void onCommit(InTxn txn) {
    // Commit handler, omit if not needed
  }
  public void onAbort(InTxn txn) {
    // Abort handler, omit if not needed
  }
}.execute();
```

Figure 10. Hyflwo2 Java compatibility API using the Atomic class.

To support open-nesting, Hyflow2 provides an Atomic abstract class with three methods: *atomically*, *onCommit* and *onAbort*. User code must subclass it and provide at least the implementation for *atomically* (see Figure 10). If implementations are provided for the other two methods, they will be used as commit and abort handlers. Unlike ScalaSTM's Java API, a transactional context object is passed to the transaction as an argument. Our reasons for doing so will become clear in Section 5.2.

5.2 Defining Hyflow2 Objects

Inheriting from a Scala trait in Java code is non-trivial. To allow a simpler way of defining Hyflow2 Objects in the Java API, we provide an abstract class called *jHObj*, which users must subclass.

Fields may be declared in two ways, which we named the Scala and the Java styles. This decision influences how the fields are later accessed from both Scala and Java code. The Scala way of declaring fields was already described in Section 3.2, and only differs cosmetically (see Figure 11). However, choosing to declare fields the Scala way makes Java code accessing that field more verbose: either the transaction context object needs to be passed explicitly to each *Ref.get* / *Ref.set* call (this object is available by sub-classing the *Atomic* abstract class as mentioned in Section 5.1), or *Ref Views* must be used to determine the context at run-time by calling *Ref.single.get* or *Ref.single.set* instead of simply *Ref.get* or *Ref.set*. The Scala style of declaring Refs is thus

```
public class Counter extends jHObj {
  Ref<Integer> value = field(0);
  public Counter() {
    Hyflow.dir().register(this);
  }

  // This method is an example transaction. It is
      not part of the Hyflow2 Object definition.
  public static void increment(final String id) {
    new Atomic {
      public void atomically(InTxn txn) {
        Counter ctr =
            Hyflow.dir().<Counter>open(id);
        // The first way of accessing Refs works
            only from an Atomic class due to the
            txn parameter
        ctr.set(ctr.get(txn) + 1, txn);
        // The second way of accessing Refs also
            works using a Runnable
        ctr.single.set(ctr.single.get() + 1);
      }
    }.execute();
} }
```

Figure 11. Scala-style Hyflow2 Object definition in Java. Notice how accessing Refs in this style is more verbose.

```
public class Counter extends jHObj {
  Ref.View<Integer> value = jfield(0);
  public Counter() {
    Hyflow.dir().register(this);
  }
  // Example transaction
  public static void increment(final String id) {
    STM.atomic(new Runnable {
      public void run() {
        Counter ctr =
            Hyflow.dir().<Counter>open(id);
        ctr.set(ctr.get() + 1);
    } } );
} }
```

Figure 12. Java-style Hyflow2 Object definition in Java. Compact Ref access.

recommended when the application is predominantly written in Scala.

For applications written mostly in Java (or even Java-only), the Java style of declaring fields makes Java code more compact. Fields are declared using *jfield* instead of *field* and their type becomes *Ref.View* instead of *Ref* (see Figure 12). Java code can now access the fields using the shorter *ref1.get()*, etc. Note that the actual method invoked is now *Ref.View.get()* and determines the transaction context object dynamically at run-time. When using the Java style, the Scala compiler will not complain if a Ref.View is accessed outside an atomic block. Instead, it would fire a single-operation transaction. Also, performance may be affected slightly due to the overheads of repeated thread-local variable lookups.

6. Mechanisms and Implementation

Our implementation uses the actor model via the Akka library.

6.1 Actors and Futures

Akka is a very efficient actor model implementation for the JVM. The actor model can lead to very fast implementations because it reduces the need for thread context switching. Actor libraries generally do their own user-space scheduling, as opposed to relying on the OS scheduler, and prohibit blocking function calls (such as disk access. etc). Instead, actors send messages to each other and respond to the messages they receive – it is an event-based programming model.

An important part of Akka's interface are *Futures*. Futures represent the result of a computation that is expected to complete at some later time. Futures can be used when a thread sends a request to an actor and expects a response. Instead of waiting for the response to arrive, the method sending the request immediately returns a Future object. The thread can register a callback to be executed when the response is received, query the Future periodically, or even block for the result. Computations can also be composed by chaining or aggregating Futures, thus reducing the number of times a thread needs to block and improving performance. Futures, as well as actors, receive and process messages and events using a configurable thread-pool.

6.2 Network Layer

Akka actors provide network transparency. They can seamlessly communicate across JVM and machine boundaries. Actor instances are identified using *ActorRef* objects. *ActorRef*s can be sent across the network while still maintaining their association with the correct actor. *ActorRef*s can then be used on the remote machine to communicate to the original actor.

Internally, Akka uses Netty for communicating over the network. Netty is a fast, asynchronous event-driven network application framework. It uses the non-blocking, high per-

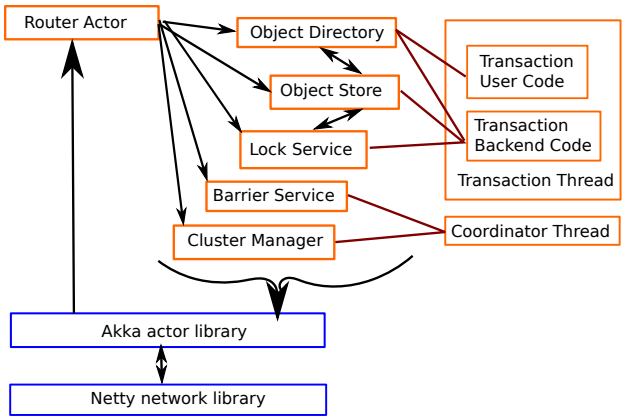


Figure 13. Hyflow2 system diagram

formance Java New I/O API. Netty also uses a configurable thread-pool for servicing received messages.

6.3 Serialization

Serialization is the process of converting an object to a format that can be sent through the network, and back. Traditionally, Java objects must implement a *Serializable* interface in order to enable this functionality. The standard Java serializer however is notorious for its weak performance. Fortunately, Akka provides an API for custom serializers, so we implemented an adapter for the Kryo library[5]. Kryo is one of the fastest JVM serialization frameworks, and is compatible with Scala.

6.4 Hyflow2 Architecture

Hyflow2 has a modular architecture. Depending on their function, module implementations need to comply to certain interfaces. Hyflow2 currently provides the following interfaces: lock service, object store, object directory, barrier service and cluster manager. A module implementation consists of a singleton object that complies to one of these interfaces and is used for sending requests to the module and an actor which services such requests. Modules communicate between each other and with the transactions' threads using message passing and Futures.

The lock service module handles acquiring, releasing and verifying the status of object and/or field locks. The object store module holds the objects themselves and handles queries, updates and validations (version checks). Due to their tight coupling, the lock service and object store can be combined in a single module. The object directory tracks object locations: it handles queries, updates, and it can also send notifications to interested transactions when an object is updated. The cluster manager tracks which nodes participate in Hyflow2 transactions, and is currently implemented by delegating a coordinator node (in the future, gossip protocols could be implemented). The barrier service lets multiple nodes coordinate their execution and is used mostly for benchmarking. An additional module is tasked with gathering statistics from all participating nodes. Figure 13 shows a system diagram which includes Hyflow2 modules and their interactions with the transaction threads and underlying libraries.

Each node has a router actor which serves as a gateway for all request messages (response messages do not pass through the gateway). The router actor dispatches messages to the appropriate module based on the message's type (Java class). This design allows every message to contain additional payload data, which can be processed in a consistent way. For example, the Transactional Forwarding Algorithm (TFA) which Hyflow2 implements needs to attach an integer (the node-local clock value) to each message sent over the network [18]. Instead of requiring every module to attach payloads to all the messages they send and receive, payloads are handled automatically in the message's base class constructor on the sender node, and is processed on the receiver node by the router actor.

6.5 Conditional Synchronization

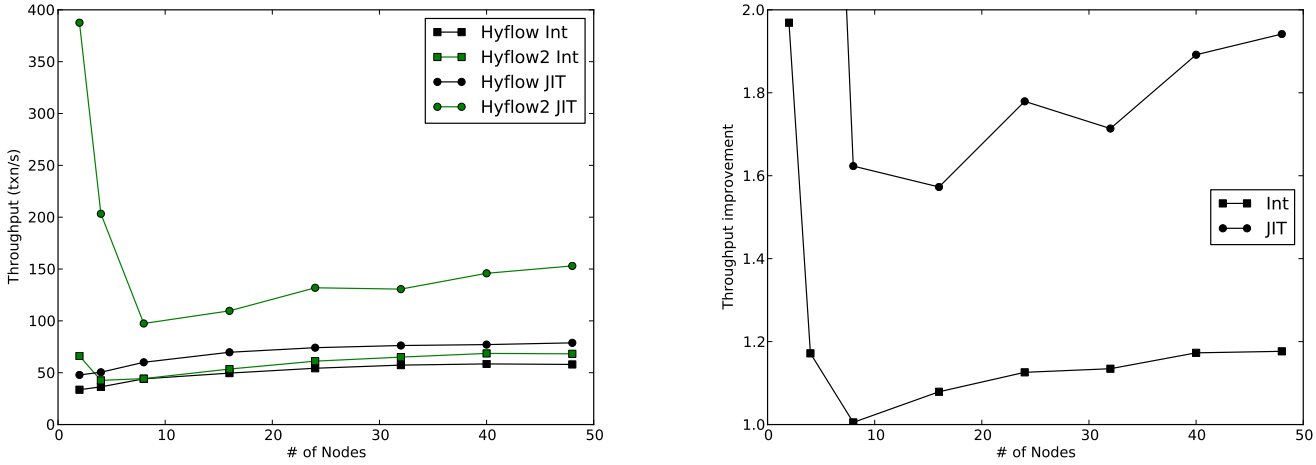
Hyflow2 is the first DTM implementation to support distributed conditional synchronization. This feature was implemented by maintaining a waiting list of transactions which are blocked on each object. When they execute, transactions record all objects they access in the transaction's read-set. When a transaction calls *retry*, it adds itself the waiting lists of all objects which it has previously read, then blocks. Waiting lists are maintained by the Object Directory. When an object is updated, the Directory is notified, and in turn notifies all transactions on that object's waiting list. Because the message adding a transaction to an object's waiting list may arrive after the object is updated, the object version is checked as well: if the transaction is waiting on an old version of the object, the notification is sent right away. Otherwise, a transaction could be waiting unnecessarily for a condition that is already satisfied.

6.6 Performance

As previously mentioned, thread context switches and network round-trip time are important bottlenecks. The choice of libraries we used in Hyflow2 was made with the purpose of addressing these issues. Akka and Netty are event-driven libraries and attempt to minimize thread context switches. We configured their internal thread pools to a minimum size that produces the greatest performance. Also, we specifically targeted serialization in our quest for performance because it is on the critical path of sending a message over the network.

7. Experimental evaluation

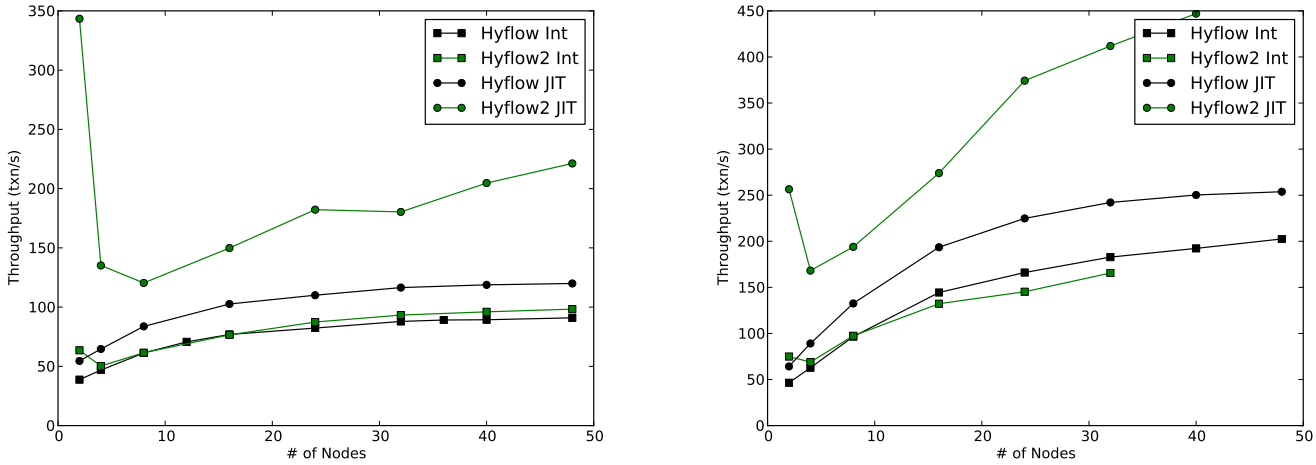
Hyflow2 was evaluated experimentally using a suite of one pseudo-macro-benchmark (bank monetary application) and three micro-benchmarks (counter and the skip-list and hash-table data structures). Since in this paper we do not seek to evaluate the TFA algorithm but rather the framework's performance, we compare against the original Hyflow which



(a) Absolute throughput

(b) Relative throughput

Figure 14. Throughput on Bank for 20% read-only transactions. 14(a) shows absolute values for both Hyflow and Hyflow2. 14(b) shows the relative improvement in Hyflow2.



(a) 50% read-only

(b) 80% read-only

Figure 15. Throughput on Bank with 50% and respectively, 80% read-only transactions.

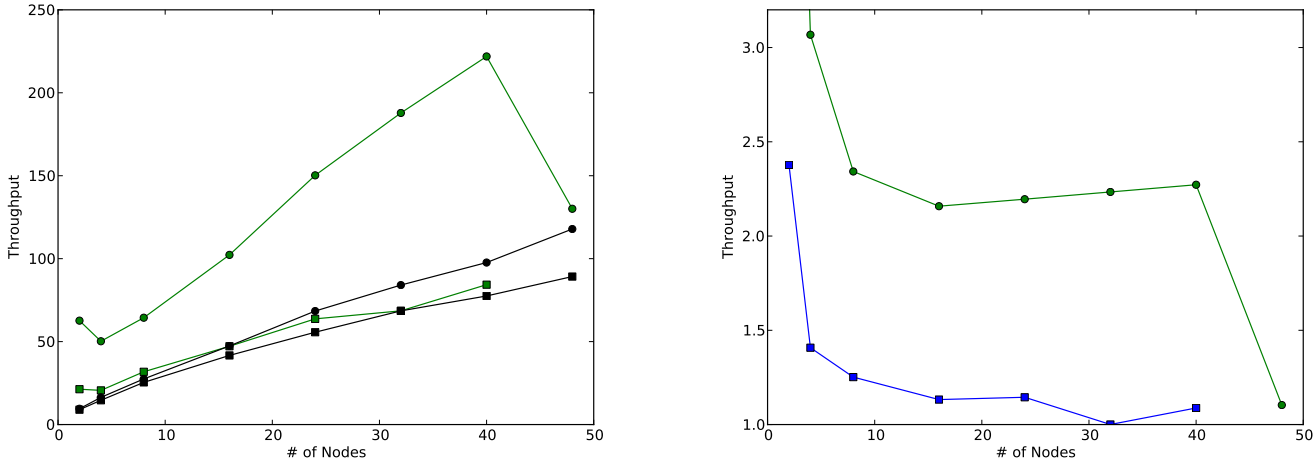
also implements TFA. Comparisons between Hyflow and other distributed transactional memory libraries implementing different algorithms are available elsewhere [18], and have shown that Hyflow outperforms competitors under most circumstances.

Experiments were run on a testbed featuring 48 identical nodes. Each node is an AMD Opteron processor running at 1700MHz. The operating system used is Ubuntu Linux 10.04 Server. Every node communicates with every other node via TCP links and the average end-to-end latency is 1*ms*. The network is not saturated.

The JVM used is the 64-bit HotSpot(TM) Server VM. Benchmarks were run first with Just-in-Time (JIT) compilation disabled (interpreted mode) and next with JIT enabled. Each test was allowed a warm-up period to compensate for compilation and class loading overheads before measurement was started.

Figure 16 shows normalized transactional throughput for each of our benchmarks. Each bar in the plot is the average of a number of measurements:

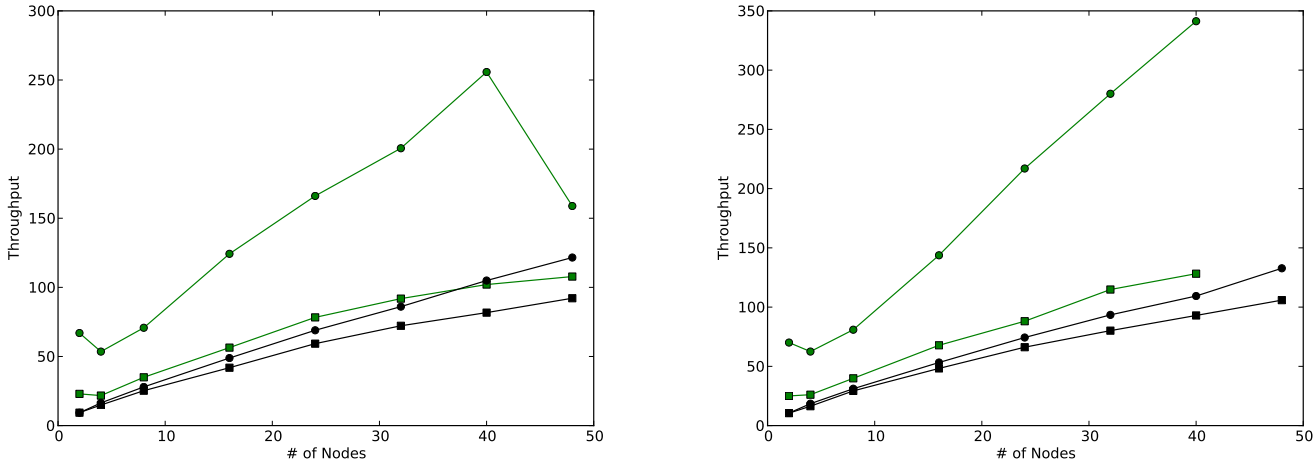
- Up to eight node count samples between two and 48 nodes.



(a) Absolute throughput

(b) Relative throughput

Figure 17. Throughput on Skip-listfor 20% read-only transactions. 17(a) shows absolute values for both Hyflow and Hyflow2. 17(b) shows the relative improvement in Hyflow2.



(a) 50% read-only

(b) 80% read-only

Figure 18. Throughput on Skip-list with 50% and respectively, 80% read-only transactions.

- Up to three contention levels determined by the amount of read-only transactions (between 0 and 80%).
- Up to three repetitions of each experiment.

We can notice that under interpreted mode, the throughput difference between Hyflow and Hyflow2 are not very significant, and vary between -20% and +25%. In compiled mode however, Hyflow2 is strikingly faster: the average speed-up is between 50% and 300%.

Figures 14 and 15 provide details on one of the benchmarks, bank. The figures follow the throughput as the number of nodes is increased from two to 48 nodes. Hyflow2 is very fast at a low number of nodes – up to 7 times faster than Hyflow with JIT enabled. When the number of nodes is in middle of the range, the improvement is only around 30-60%. Then, as more nodes are added, Hyflow2's performance benefit keeps steadily increasing up to just bellow 100%. When JIT is disabled the trends are similar, but improvements are only in the 0-20% range (and even negative in limited cases).

Figures 17 and 18 show the same trends for the Skip-list benchmark.

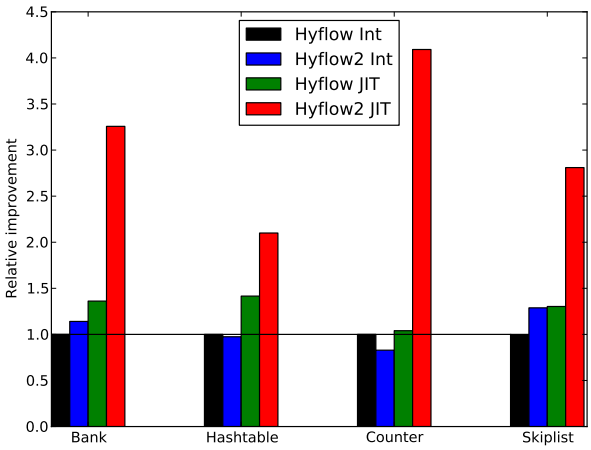


Figure 16. Summary of relative performance across benchmarks.

8. Related Work

DecentSTM [8] is a decentralized STM algorithm providing the snapshot isolation consistency guarantee. The reference implementation provided by the authors does not function in a real distributed setting, but rather emulates it using threads.

GenRSTM [10] uses group communication services to implement a distributed STM. Its API uses Box containers, not unlike Hyflow2's Refs. GenRSTM is modular and can be used to implement multiple STM algorithms.

Both these competitor DTM frameworks were compared against Hyflow in [18].

9. Conclusion

We introduced Hyflow2, a high performance distributed transactional memory for the JVM. Hyow2 is the first Distributed Transactional Memory implementation with support for Scala, interoperability with Java, and key DTM features including nested transactions and distributed conditional synchronization. We focused on performance, and managed to significantly improve transactional throughput compared to the original Hyflow. Future work may include support for checkpointing as an alternative to closed nesting, configurable field/object level locking and alternative atomic blocks with distributed selective waiting.

References

[1] Akka (toolkit and runtime for building highly concurrent, distributed and fault tolerant event-driven applications on the jvm), April 2012. URL `http://akka.io/`.

[2] Ehcache (java distributed cache), April 2012. URL `http://ehcache.org/`.

[3] Hazelcast (data distribution platform for java). http://www.hazelcast.com/, April 2012. URL `http://www.hazelcast.com/`.

[4] Jboss infinispan (distributed data-grid platform), April 2012. URL `http://www.jboss.org/infinispan`.

[5] Kryo (jvm serialization library), April 2012. URL `http://code.google.com/p/kryo/`.

[6] Scalastm (software transactional memory api for scala), April 2012. URL `http://nbronson.github.com/scala-stm/`.

[7] Apache zookeeper (distributed configuration service, synchronization service and naming registry), April 2012. URL `http://zookeeper.apache.org/`.

[8] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized stm algorithm. In *IPDPS*, pages 1–12. IEEE, 2010.

[9] N. G. Bronson, H. Chafi, and K. Olukotun. Ccstm: A library-based stm for scala. The First Annual Scala Workshop at Scala Days 2010, April 2010.

[10] N. Carvalho, P. Romano, and L. Rodrigues. A generic framework for replicated software transactional memories. In *NCA*, pages 271–274. IEEE Computer Society, 2011. ISBN 978-1-4577-1052-0.

[11] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[12] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In P. Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2005. ISBN 3-540-29163-6.

[13] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL `http://doi.acm.org/10.1145/359576.359585`.

[14] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[15] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In K. A. Yelick and J. M. Mellor-Crummey, editors, *PPOPP*, pages 68–78. ACM, 2007. ISBN 978-1-59593-602-8.

[16] M. M. Saad. Hyflow: A high performance distributed software transactional memory framework. Master's thesis, Virginia Tech, April 2011.

[17] M. M. Saad and B. Ravindran. Supporting stm in distributed systems: Mechanisms and a java framework. In *TRANSACT (ACM SIGPLAN Workshop on Transactional Computing)*, San Jose, California, USA, June 2011.

[18] M. M. Saad and B. Ravindran. Transactional forwarding algorithm. Technical report, Virginia Tech, January 2012.

[19] A. Turcu and B. Ravindran. Hyflow2: A high performance distributed transactional memory framework in scala. Technical report, Virginia Tech, April 2012. URL `http://hyflow.org/hyflow/chrome/site/pub/hyflow2-tech.pdf`.

[20] A. Turcu and B. Ravindran. On open nesting in distributed transactional memory. In *SYSTOR 12*, Haifa, Israel, June 2012. URL

http://hyflow.org/hyflow/chrome/site/pub/opennesting-systor12-tech.pdf.

[21] A. Turcu, B. Ravindran, and M. M. Saad. On closed nesting in distributed transactional memory. In *TRANSACT*, 2012.