Hyflow2: A High Performance Distributed Transactional Memory Framework in Scala

Full research paper

Alexandru Turcu  
Virginia Tech  
talex@vt.edu

Binoy Ravindran  
Virginia Tech  
binoy@vt.edu

Roberto Palmieri  
Virginia Tech  
robertop@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 known as the control-flow model, because the computation moves where the data is). Synchronization primitives are available using dedicated platforms like Apache Zookeeper [31] and Hazelcast [27] or can be implemented ad-hoc. Alternatively, in the data-flow model, distributed caches such as Ehcache [26] and Infinispan [28] 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, deadlocks 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 [9], 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, 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

[Copyright notice will appear here once 'preprint' option is removed.]
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 [8] 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/.

In designing Hyflow2 we focused on several issues that could be improved compared to the original Hyflow [18]: 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 [25] 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 ScalaSTAM API [2, 30], 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) [19, 20], a DTM technique that uses the data-flow model (immobile transactions, mobile objects). We support the flat, closed and open nesting models [22, 23], as well as distributed conditional synchronization.

To the best of our knowledge, Hyflow2 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 overviews TFA, the protocol implemented in Hyflow and Hyflow2. Section 3 briefly describes the original Hyflow library and the areas where it was lacking. Section 4 introduces Hyflow2’s new API. Section 5 describes transactional nesting and the API for supporting it in Hyflow2. Implementation is discussed in Section 7. Hyflow2 is experimentally evaluated in Section 8. Related work is briefly mentioned in Section 9 and Section 10 concludes the paper.

2. Overview of TFA

For completeness, we overview TFA. TFA [19, 20] is based on the TL2 algorithm, already proposed for multiprocessor TM [6]. It is a data flow based, distributed transaction management algorithm, which provides atomicity, consistency, and isolation properties. Under TFA, operations on distributed objects are buffered and locks on objects are acquired at commit time. On successful acquisition of locks, objects are updated. Otherwise, the transaction is aborted by releasing all previously acquired locks and retried.

In contrast to TL2’s central clock, TFA uses independent, per-node transactional clocks and provides a mechanism to establish the “happens before” relationship between significant events (e.g., write-after-write, read-after-write). Upon a transaction’s successful commit, a node increments its local clock. An object’s version is defined by the local clock at the time of the object’s last modification. When a local object is accessed by a transaction, as part of validation, the object version is compared with the transaction’s starting time. If the object’s version is newer, the transaction is aborted and retried.

For validating remote objects, TFA employs a technique called “transaction forwarding:” when a transaction requests access to a remote object, the local clock is piggybacked with the request to the remote node. The remote node advances its clock to the sender’s clock if its clock is older; otherwise, no update is made to the remote clock.

The remote node then sends the object copy with its clock value. Upon receipt, the local node (i.e., the sender) compares the remote clock value with the transaction starting time. If the remote clock is newer, the transaction’s read-set is validated by checking whether any other object in the read-set has been updated to a version newer than the transaction starting time. If the read-set validation succeeds, then the transaction starting time is advanced to the remote clock value (i.e., “forwarded”). Otherwise, the transaction is aborted and re-issued.

When a transaction reaches the commit stage, it first acquires locks on all the objects in its write-set. On successful lock acquisition, the objects are updated and the transaction committing node is published as the new host of the updated objects. If lock acquisition fails for any object, all acquired locks are released, and the transaction is aborted and re-issued.

2.1 Illustrative Example

Consider three nodes, N1, N2, and N3, each running transactions T1, T2, and T3, respectively (see Figure 1). N2 hosts object θ and is considered θ’s “owner.”

Nodes maintain a local transactional clock, which is incremented when transactions running on them commit. T3 starts at local clock lc3 = 12 and requests θ from N2 at lc3 = 16. N3 compares the received clock value with its local clock lc2 = 12 and advances its clock to lc2 = 16. After some time, T1 starts at local clock lc1 = 14 and requests θ from N2 at lc1 = 19. At N2, no change is made to its local clock lc2, since rc = 19 < lc2 = 21.

When N1 receives the response from N2, it observes that lc1 = 19 < rc = 21. Therefore, it forwards T1 to start at lc1 = 21 and validates all other objects accessed earlier by T1. Later, T2 acquires the lock on θ at N3 and updates θ at local clock lc3 = 25. Now, N3 holds the ownership of θ, but leaves θ locked at N2. When T1 tries to acquire the lock on θ, it may find N2 or N3 as the object owner depending on when N3 successfully publishes its ownership of θ. If T1 requests the lock on θ from N2, it fails due to the existing lock on θ and aborts. Otherwise, if T1 finds N3 as the owner, it acquires the lock, but fails during read-set validation. Therefore, T1 releases the lock acquired on θ at N3 and aborts. T2 retries and requests θ again from the new owner N3.

Concurrently, another transaction T2 also receives θ, but T1 acquires the lock on θ earlier than T2 and commits. As a result, T2 aborts and retries. T2 finally commits at lc2 = 44 and becomes the new object owner.

3. The Hyflow DTM framework

Hyflow is the original DTM prototype implementing TFA [18]. It was built on top of the Deuce STM library and the Aleph communication framework. Hyflow’s modular design attempts to allow for pluggable network transports, transactional algorithms, directory protocols and contention managers. However its interfaces were not abstract enough to allow the implementation of more complex algorithms and any possible work around resulted in a source code difficult to maintain.

Hyflow (just like the underlying Deuce STM) relies on automatic byte-code rewriting to provide an API based on annotations. As seen in Figure 2, 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
Hyflow2: A High Performance Distributed Transactional Memory Framework in Scala

Hyflow2 is focused and optimized addressing the real distributed systems bottlenecks, namely the network round-trip time and thread context switch overheads (details are presented in Section 7.8).

4. Hyflow2 API

Hyflow2 API is based on the excellent ScalaSTM API[30]. In fact, Hyflow2 tries to reuse ScalaSTM’s interfaces wherever possible, and partially implements a back-end for the ScalaSTM API.

4.1 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[2]. Hyflow2 inherits all features described in this section.

Transactions in ScalaSTM are defined using atomic blocks, as shown in Figure 3. To achieve this syntax, atomic 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 (Refs) 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 transactional reference (Ref), one would use

4. Hyflow2 API

Hyflow2 API is based on the excellent ScalaSTM API[30]. In fact, Hyflow2 tries to reuse ScalaSTM’s interfaces wherever possible, and partially implements a back-end for the ScalaSTM API.

4.1 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[2]. Hyflow2 inherits all features described in this section.

Transactions in ScalaSTM are defined using atomic blocks, as shown in Figure 3. To achieve this syntax, atomic 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 (Refs) 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 transactional reference (Ref), one would use

Figure 4. A more verbose version of the code in Figure 3, with several Scala syntactic shortcuts written explicitly.

Figure 3. An example transaction in ScalaSTM (common usage).

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

Figure 2. Example of the original Hyflow API. Transactions are marked using the @Atomic annotation.

@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 transaction execution under TFA.

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 the instrumented byte-code cannot readily be debugged, 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.

Conversely, Hyflow2 is focused and optimized addressing the real distributed systems bottlenecks, namely the network round-trip

time and thread context switch overheads (details are presented in Section 7.8).
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 5. 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 value = field(0) // an integer field
    private var val = field(0) // an integer field
    override protected def update(): InTxn = atomic {
        // update
        this.head() = old_head.next
        return old_head.value
    }
}

// Figure 6. Hyflow2 Object example for a bank account.

// 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 3, the txn object is automatically passed to the apply() and
// update() methods. Figure 4 shows how Scala interprets the code in
// Figure 3.

// This mechanism using implicit arguments and Refs leads to a
// clean syntax with relatively little redundant code (only the "implicit
// txn = $\Rightarrow\) construct and the function call "(" characters are super-
// fluous). Another benefit of this mechanism is protecting against
// concurrent access of a memory location from both transactional
// code and non-transactional code. This property is highly desir-
// able in TM systems because in such scenarios, the behavior of
// interleaving transactional with non-transactional operations is unde-
// fined. 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 however, no transaction context value is implicitly
// available, so calls to apply or update would lead to compila-
// tion 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 5).
// When the queue is empty, this operation may invoke retry, effec-
// tively blocking until at least one element is available. This behavior
// is called conditional synchronization. After calling retry, the trans-
// action 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.

4.2 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 6). An Hyflow2 Object (henceforth referred
to as HObj) mixes in the HObj Scala trait \(^1\) and it represents
the Hyflow2’s basic unit of data. Each Hyflow2 Object 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.

4.3 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 6). 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 open-
ing (see Figure 7). It requires the identifier of the requested object
and it generally caches a copy of the requested object on the local
node.

5. Transaction Nesting

Hyflow2 includes support for nested atomic blocks. In this section
we first briefly describe the three nesting models previously studied
in TM \(\cite{7,12}\): flat, closed and open. Subsequently we introduce the
API support for nesting in Hyflow2, and explain its use. Lastly, we
make the case for a third atomic construct.

5.1 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.

Open nesting

In open nesting, operations are considered at a higher level

\(^1\) 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.
of abstraction. Open-nested transactions are allowed to make
their changes visible and commit to the shared memory inde-
pendently 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 re-
vert 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. Al-
though 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.

5.2 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 expose the decision of which of the two models to use in the
interchangeably. Unlike in the original Hyflow, we decided not
exposing the differences between the two models. Currently, the
decision is fixed based on a configuration value, but in the future it could be made
adaptively at runtime.

Open nesting on the other hand requires API support. Following
the style of ScalaSTM, in Hyflow2 we propose the following syntax
(see Figure 8):

- An open nested transaction should be started with \texttt{atomic.open}.
The body of the transaction follows the syntax of regular trans-
actions.

- Following the transaction’s body two optional blocks may be
specified. These blocks are introduced by \texttt{onCommit} and \texttt{onAbort}, and represent the transaction’s commit and abort han-
dlers, respectively. The handlers themselves are executed as
open-nested transactions, so they must accept the implicit trans-
action context argument. If both handlers are present, their or-
der 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 \texttt{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 8).

- For any other abstract lock scenarios, the locks must be ac-
quired within the sub-transaction’s body using \texttt{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, \texttt{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.

5.3 Discussion and Language Mechanisms

We consider \texttt{atomic.open} a semantically cleaner way of denot-
ing open-nesting transactions than the previously suggested open-
atomic keyword [13]. Our syntax logically breaks down into two
terms. The first term, \texttt{atomic} is the same as the marker for regular
atomic blocks. The second term, \texttt{open}, appears as a property of the
resulting transaction. By contrast, \texttt{opennatmetic} as a separate key-
word, gives the impression the effect is totally unrelated with that
of the \texttt{atomic} keyword.

When evaluating an \texttt{atomic.open} block, the open method is
called on the \texttt{atomic} object of type \texttt{TxnExecutor}, and it receives
the function to be executed transactionally as a parameter. Declaring
the \texttt{onCommit} and \texttt{onAbort} handlers is more complex: blocks are
evaluated last to first, wrapping what is above in a special
\texttt{OpenNestingBlock} container object, and calling \texttt{onCommit/onAbort}
on this object. The object is saved in a thread-local variable. When
finally, \texttt{atomic.open} is invoked, it checks if there is any \texttt{Open-
NestingBlock} object registered for the current thread and uses it, if
any. See Figure 9 for an expanded example. This mechanism is also
used in ScalaSTM to implement the \texttt{orElse} keyword (\texttt{orElse}
provides the means to execute alternative atomic blocks if the original
ones fail).

5.4 Configurable nesting

For testing reasons users may need to execute certain atomic blocks
under both flat/closed and open nesting models. Using the pre-
viously described API, switching between models would require
modifying the source code and recompiling. To avoid this situa-
tion, we support an additional method for launching a transaction,
"\texttt{atomic.config}". An atomic block marked with \texttt{atomic.config} will
determine its nesting model at run-time, by reading it from a config-
uration value. For completeness, the choice between flat and closed
nesting is explicit. \texttt{Atomic.config} allows defining abort and commit

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure8.png}
\caption{Open nesting in Hyflow2}
\end{figure}
handlers just like `atomic.open`. If the block executes with closed or flat nesting, these handlers will simply be ignored.

6. 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`, `Hyflow.dir().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.

6.1 Defining Transactions

ScalaSTM already provides a way for starting transactions from Java code, so we need to provide additional mechanisms to obtain the same results.

6.2 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.

```
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 run() {
                Counter ctr = Hyflow.dir().<Counter>open(id);
                ctr.set(ctr.get() + 1);
            }
        }.execute();
    }

    // Example transaction
    public static void increment(final String id) {
        new Atomic {
            public void run() {
                Counter ctr = Hyflow.dir().<Counter>open(id);
                ctr.set(ctr.get() + 1, ctr); // The second way of accessing Refs also works using a Runnable
                ctr.single.set(ctr.single.get() + 1);
            }
        }.execute();
    }
}
```

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

For applications written mostly in Java (or even Java-only), the Java style of declaring fields 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 6.1), 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 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 `RefView` instead of `Ref` (see Figure 13). Java code can now access the fields using the shorter `ref1.get()`, etc. Note that the actual method invoked is now `RefView.get()` and determines the transaction context object dynamically at run-time. When using the Java style, the Scala compiler will not complain if a `RefView` 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.

```
public class Counter extends jHObj {
    public void onAbort(InTxn txn) {
        ctr.value.set(ctr.value.get() + 1);
        System.out.println(ctr.value.get());
    }

    public void onCommit(InTxn txn) {
        System.out.println(ctr.value.get());
    }

    public Boolean atomically(InTxn txn) {
        return true;
    }

    public void run() {
        ctr.set(ctr.get() + 1);
        ctr.set(ctr.get(txn) + 1, txn);
    }
}
```

Figure 12. Scala-style Hyflow2 Object definition in Java. Notice how accessing Refs in this style is more verbose.

```
new Atomic<Boolean> {
    public Boolean atomically(InTxn txn) {
        Counter ctr = Hyflow.dir().<Counter>open("ctr");
        ctr.set(ctr.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 11. Hyflow2 Java compatibility API using the Atomic class.

```
public class Counter {
    Ref<Integer> value = field(0);

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

Figure 13. Java-style Hyflow2 Object definition in Java. Compact Ref access.

```java
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 10. ScalaSTM Java compatibility API.
7. Mechanisms and Implementation

Our implementation uses the actor model via the Akka library.

7.1 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.

7.2 Network Layer

Akka actors provide network transparency. They can seamlessly communicate across JVM and machine boundaries. Actor instances are identified using ActorRef objects. ActorRefs can be sent across the network while still maintaining their association with the correct actor. ActorRefs 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 performance Java New I/O API. Netty also uses a configurable thread-pool for servicing received messages.

7.3 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[29]. Kryo is one of the fastest JVM serialization frameworks, and is compatible with Scala.

7.4 Hyflow2 Architecture

Hyflow2 has a modular architecture. Depending on its 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 implements a Lock interface 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 (see Section 2). 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.

7.5 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.

7.6 Parallel Object Open

This is another feature provided by Hyflow2 that can speed up certain transactions. Since objects are usually retrieved from remote nodes, the open operation is time-consuming. When a transaction needs multiple objects and knows their identity in advance, it can use the parallel object open operation to reduce the number of

---

The diagram shown in Figure 14 illustrates the Hyflow2 system architecture, highlighting the roles and interactions of various components such as the Router Actor, Object Directory, Transaction Code, and cluster manager, among others. This visual representation aids in understanding the modular design and interplay of different modules within the Hyflow2 system.
work round-trips required for acquiring a copy for each required object.

7.7 Transaction Checkpoints
Checkpoints were proposed by Koskinen and Herlihy [11] as an alternate mechanism for partial rollback. As opposed to nesting, where execution can return only to sub-transaction boundaries, checkpoints allow resuming execution from any desired location where a checkpoint was saved. Checkpoints rely on Continuations, a programming language mechanism that allow saving and resuming the control state of a program. At their core, continuations work by saving and restoring the CPU registers and the activation stack.

While some languages have varying degrees of support for continuations (e.g. in C one could use setcontext/setcontext or setjmp/longjmp), the official Java Virtual Machine does not support this feature. In order to add support for continuations in Java, a number of paths are available:

- Use a library that employs byte-code rewriting, such as JavaFlow, NightWolf or, with modifications, Kilim. Such libraries employ a user-code level activation stack (as opposed to a JVM-level stack) and modify all local variable accesses to explicitly use this stack.
- Use an alternative JVM with support for continuations, such as the Avian JVM.
- Modify the the open-source JVM to support continuations. A patch is available for this purpose in the Da Vinci Machine Project.

For Hyflow2 we chose the third approach, as it gives the best performance. While this requires a non-standard JVM, Hyflow2 can run on stock JVM with checkpoints disabled.

7.8 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.

8. Experimental evaluation
Hyflow2 was evaluated experimentally using a suite of:

- Bank benchmark, a benchmark that mimics a monetary bank application, widely used for evaluating STM and DTM systems [4, 5, 24];
- Enhanced Counter, Skip-List and Hash-Table, three micro-benchmarks typically used for stressing TM systems [15, 22, 23]. In the first, transactions access counter objects which they read or increment; the other two are configurable applications acting on distributed data structures, respectively skip-list and hash-table.

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 also implements TFA. Comparisons between Hyflow2 and other distributed transactional memory libraries implementing different algorithms are available elsewhere [19, 20], 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 1ms. 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 15. Summary of relative performance across benchmarks.](image-url)

Figure 15 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;
- 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 16 and 17 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 limited to 20%.

Figures 18 and 19 show the same trends for the Skip-List benchmark.

9. Related Work
DecentSTM [1] 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 [3] 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 [19, 20].

In context of DTM, a number of papers recently appeared [14, 16, 17, 21]. They provide new protocols optimizing peculiarity of different scenarios and all are based on control-flow, without implementing a directory based protocol for looking-up shared objects among nodes. These protocols cannot be compared with our framework because it provides the implementation of a data-flow based protocol.

Hyflow2 has been recently used as a reference DTM framework in [10].

10. 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.

Acknowledgment

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

References

(a) Absolute throughput

(b) Relative throughput

Figure 18. Throughput on Skip-List for 20% read-only transactions. 18(a) shows absolute values for both Hyflow and Hyflow2. 18(b) shows the relative improvement in Hyflow2.

(10% read-only)

(50% read-only)

(80% read-only)

Figure 19. Throughput on Skip-List with 50% and respectively, 80% read-only transactions.


Hyflow2: A High Performance Distributed Transactional Memory Framework in Scala

2013/7/1