Streamline

Streamline is a programing framework for event driven, high throughput systems. The design is based on a Software Transactional Memory, Immutable data structures and a propriatary implementation of data objects. Events are processed on one single thread from the spawning of the event until the last consequence of that event has been processed. Data is updated via Transactions that are ACID compiant. The goal of Streamline is to create a framwork for highly concurrent event processing without the need for context switches or messaging.

Benefits

No need for external synchronization.

No Need for expensive Context switching.

Priority inversion, makes it possible to enhance throughput for important events on behalf of less important Events

Message passing functionallity is a natural part of MVCC. Other system may implement the handling of IProducedData as messages.

No copying of data. As long as chunks of data is represented by an IProducedData, fileds does not need to be copyed into other object, just referenced.

Unlimited scaling on increasing number of processor cores.

Compliant with Amdahls law.

Drawbacks

Overhead in field assignment.

Overhead in Transaction based updates.

Running codepaths in a multi-threaded environment compared to a single traded environment may lead to unoptimized processor caching strategies.

When to use Streamline

Streamline is designed to adress the challenges of event driven soft Realtime systems that require high throuhput. Streamline is most suitable in environments where multiple event driven objects have a complex set of dependencies between one and other. E.g:

Services to handle financial products and pricing

Exchanges

Betting services

Business intelligence services

Real time data analysis services

When complex arithmetic and state transitions is preventing a high throughput system to run in a single thread context, Streamline is providing a solution to scale the arithmetic operations over multiple threads with only a small synchronization footprint.

When not to use Streamline

In environments where the number of field assignments are large and arithmetic processing is low, Streamline's filed assignment overhead is likely to cost more then the scalability gain. If the throughput can be met in a single thread context it is likely to be the preferred choice of design pattern.

In real life it is likely to come across requirements where part of a system, a service or work unit can handle its throughput within a single thread context, but other services or work units needs to scale over multiple processor cores. It is possible to combine a more classic producer/comsumer pattern in combination with the Streamline pattern as a IDatasProducerService may run its events in a single thread context and post IProduced Data as messages into the STM to reach its subscribers. this approach, like any producer/consumer implementation will need to use context switches at the cost of both latency and throughput.

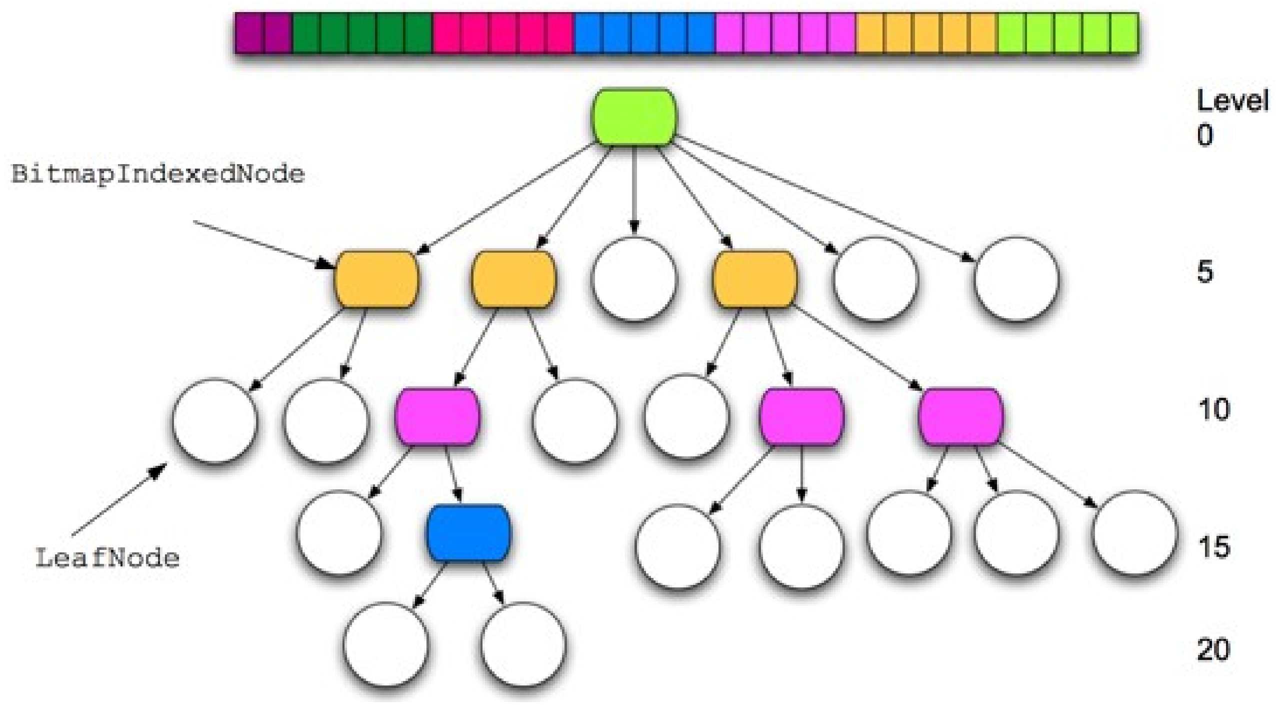
DataType

Data types are considered raw data in Streamline. In addition to Java datatypes DataType encapulates data type characteristics under a common super class and contains serialization methods.

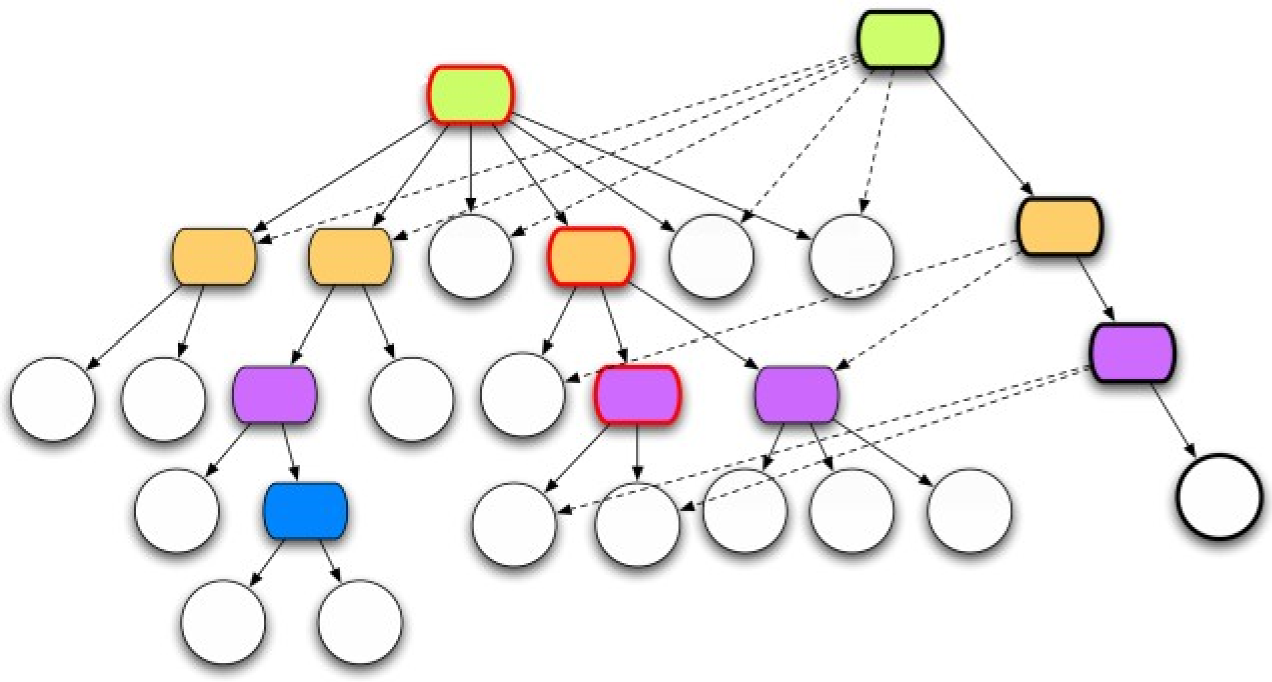
Persistent HashMap

PersistentHashMap is an implementation of bitmap indexed hashmap created by Rick Hickey for JVM based lisp Clojure. The beauty of this implementation is that it is really fast and that the map is immutable. Through the use of data structure sharing between versions of the map, a cheep copy is made with little overhead. In Clojure the concept of creating cheap immutable copies upon writes is called Multi Version Concurrency Control.

Bitmap indexed HashMap



DataStructure sharing:



Read more about this datastructure at:

http://blog.higher-order.net/2009/09/08/understanding-clojures-persistenthashmap-deftwice/

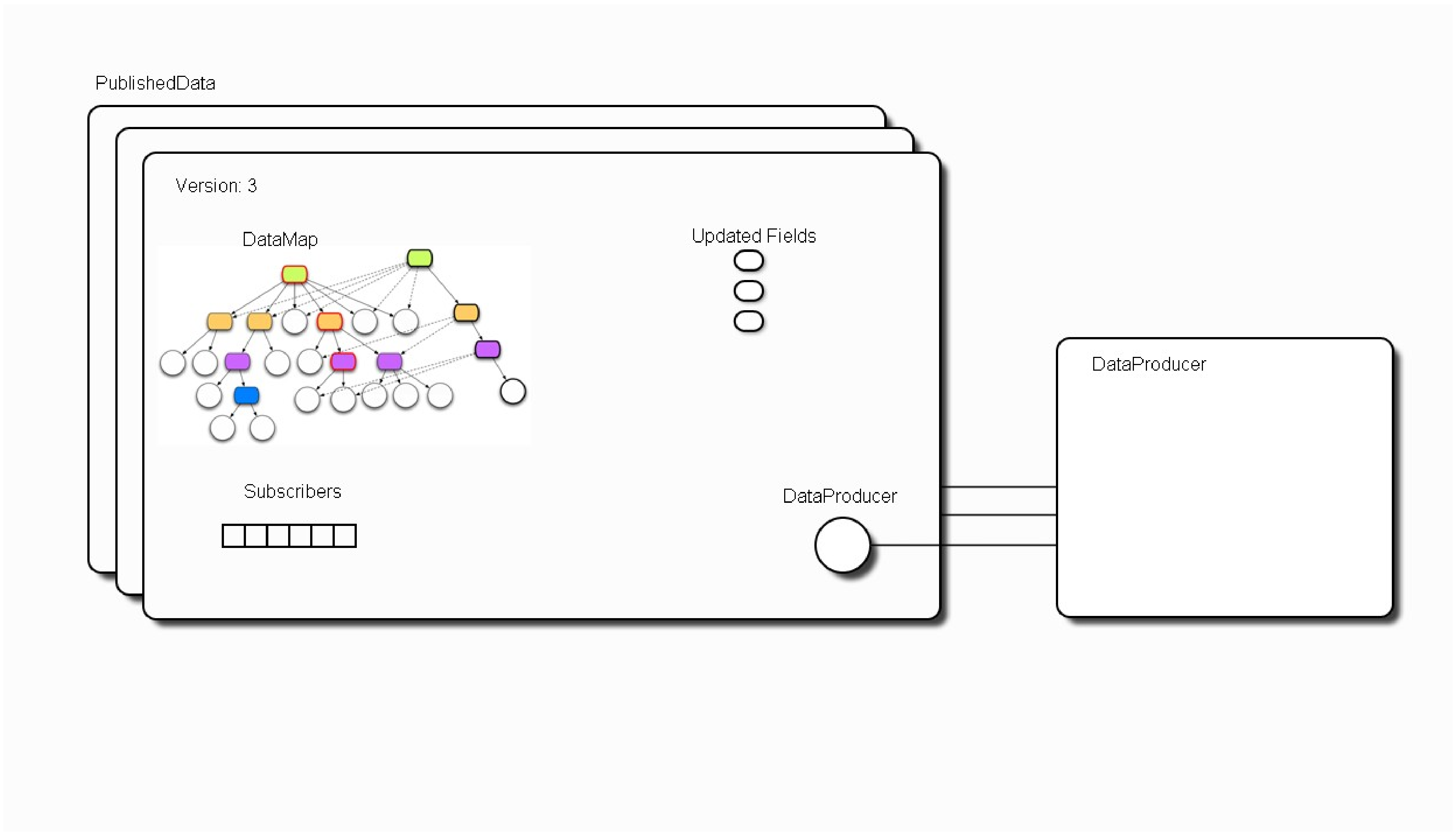
IPublishedData

IPublishedData is the key elements of Streamline. Every exposed entity of the Streamline application should be considered a publishedData. A PublishedData contains the last DataType values in a PersistentHashMap, a version number, an integer set containng the field ID's of the last updated valus, a subscriber list and a reference to its producer. A publishedData is immutable and represents the concurrent state of the entity. A PublishedData can only be updated to a new version via the STM or via its producer. The STM will update the PublishedData for standard instructions such as setting default status or adding a subscriber. A PublishedData is created by the STM whenever a subscriber requests a subscription for a DataKey. Every DataKey is unique and is either retrieved by a key constant or requested asynchronously from a DataProducerService. It is the DataProducerService defined within the DataKey to provide the appropriate DataProducer for each PublishedData

DataProducer

The Data producer is initiated upon the creation of a PublishedData. Every published data must be initialized with a producer. The producer will be active during the lifetime of its PublishedData and after the last subscriber has left the PublishedData and left it in a “of no one's interest” state, the published data will be cleaned out and the producer will be stopped and discarded. It is the responsibillity of the DataProducer to update the PublishedData. This is done via transactions that are created by the Publisher and executed by the STM. The version number of the producer is validated by the STM, so updates from discarded producers that tries to updated a newly created version of its data key will affectively be discarded.

Anatomy and relationship, Published data and Data Producer



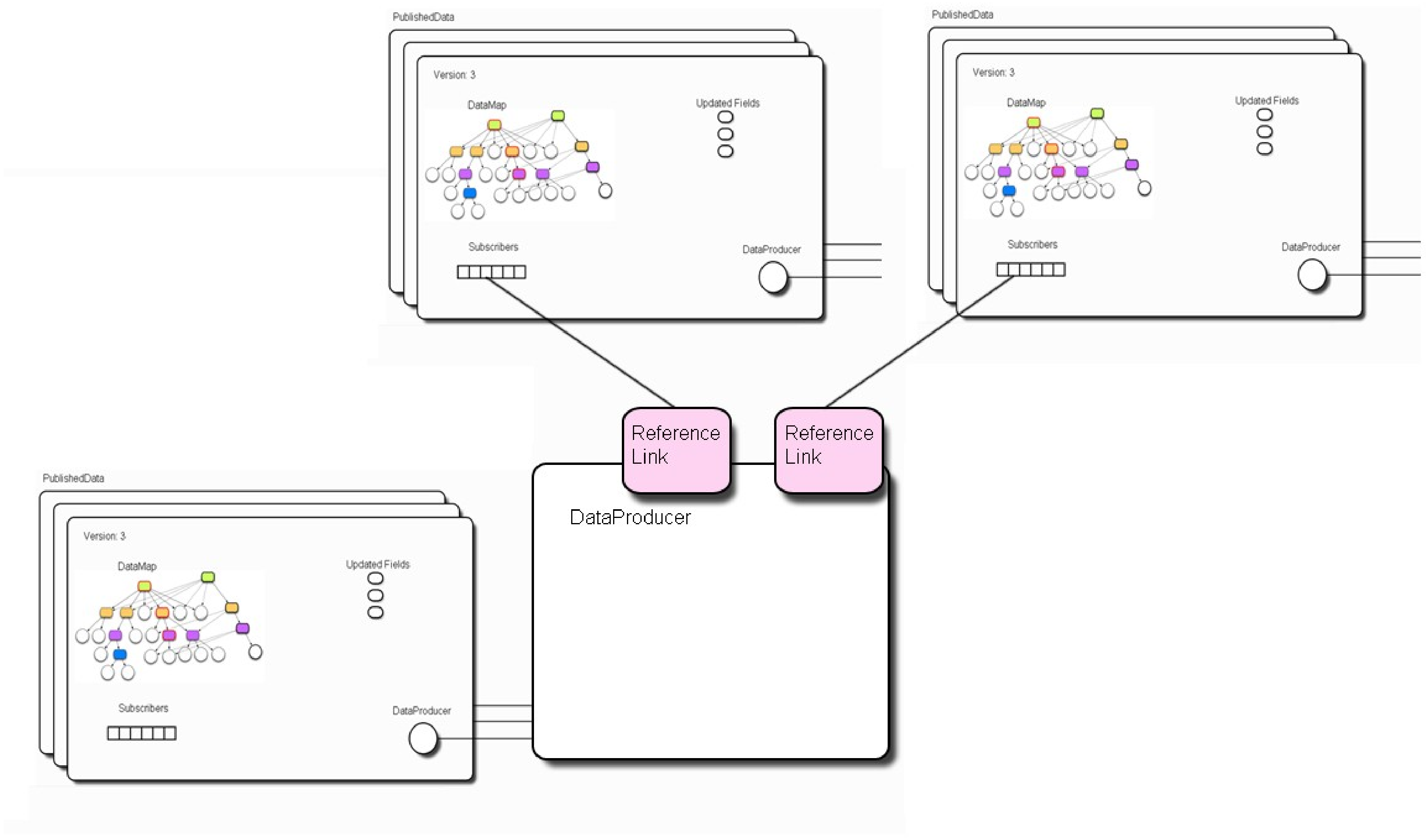
TemporaryController

A temporary controller is an object with an internal synchronize mechanism for start/stop method. As the state transition of the IPublishedData is done in an atomic fashion, the consequences of the state change will take place outside the transaction and hence be exposed to the harsh world of concurrent events. These controllers are meant to handle connections to external sources that may affect and alter the state of the PublishedData. The DataProducer is a temporary controller. Also linkes to other Entries in the STM (other IpublishedData) are of the type Temporary controller.

DataTypeLink & ReferenceLink

The datatype link represents a link to another STM entry. If within a transaction a programmer decides to add field with a link value, after the transaction is completed a reference link will be applied to the entry's producer. That link will begin to receive updates from the subscribed entry. Every update will end up in a new transaction beeing executed to also update the subscriber state so that the field always points to the last version of the subscribed entry. It is possible to intercept the transaction of the reference update by overriding the "`referenceDataCall( inFieldKey, inLink, inData, this )`" in the Producer. This allows the programmer to inspect the update and do more stuff within the same transaction. There is also the possibility to intercept the reference call after the transaction has been completed and executed by the STM by overriding the `postReferenceDataCall( inFieldKey, inLink, inData ) method.` ReferenceLink extends TemporaryController.

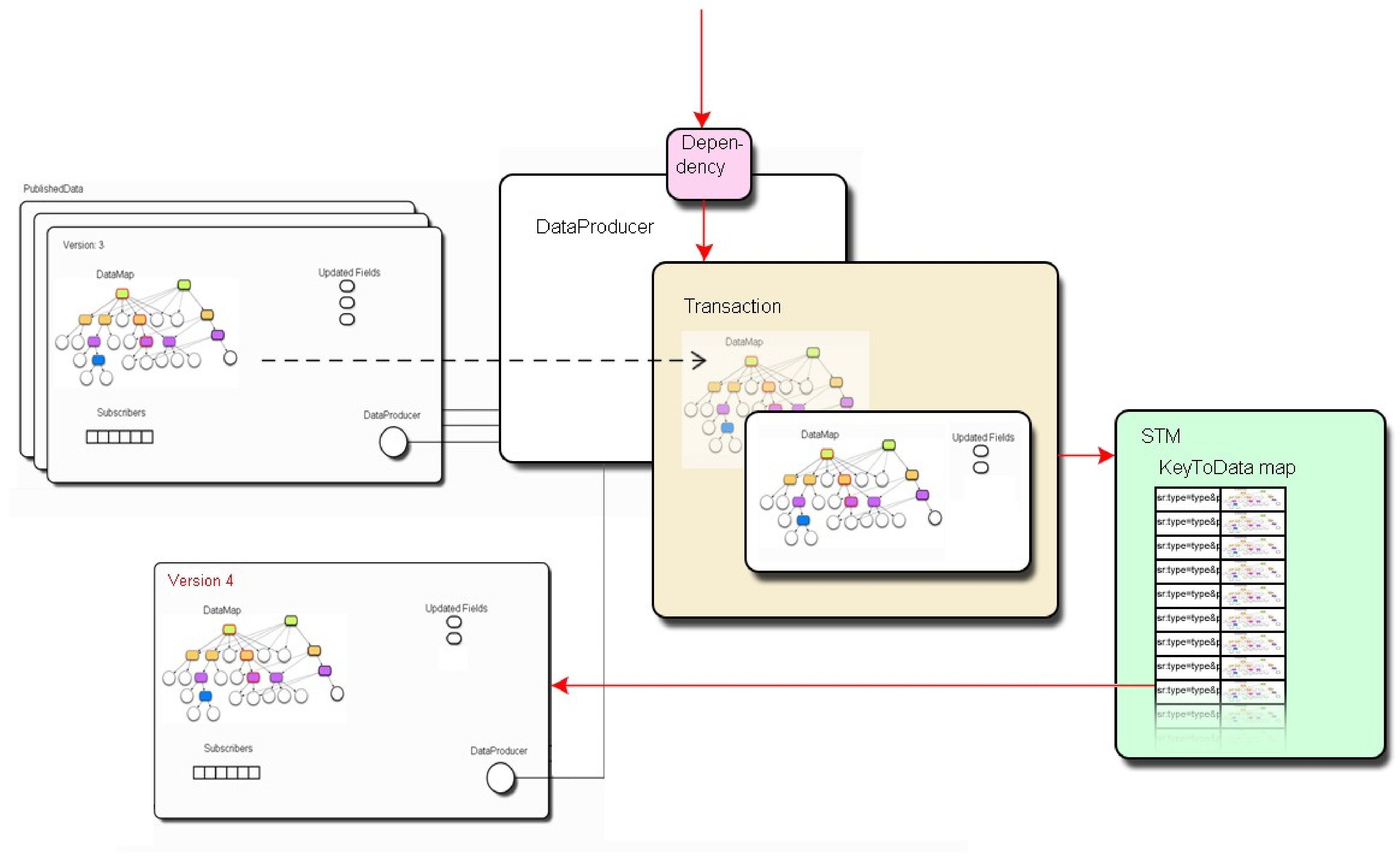
Linkage of STM entries in a pub/sub fashion via DataTypeLink and ReferenceLink



STM event processing.

This picture below represents the process of updating an STM entry.

The red arrow coming in from the top is an event from external source (possibly via a reference link to another STM entry). This event is captured by a temporaryController to the DataProducer. The producer creates a transaction to modify its entry according to the new information. Within the transaction the state transition code is written, and the transaction is dispatched to the STM. The STM initiates its locking mechanisms and runs the transaction. When the transaction is complete, the old version is atomically replaced by the new version in the memory. The synchronized execution is then complete. The remaining work for the STM is the deal with the consequences of the state transition. TemporaryControllers and Reference links etc. is set up and started.



DataSequencer

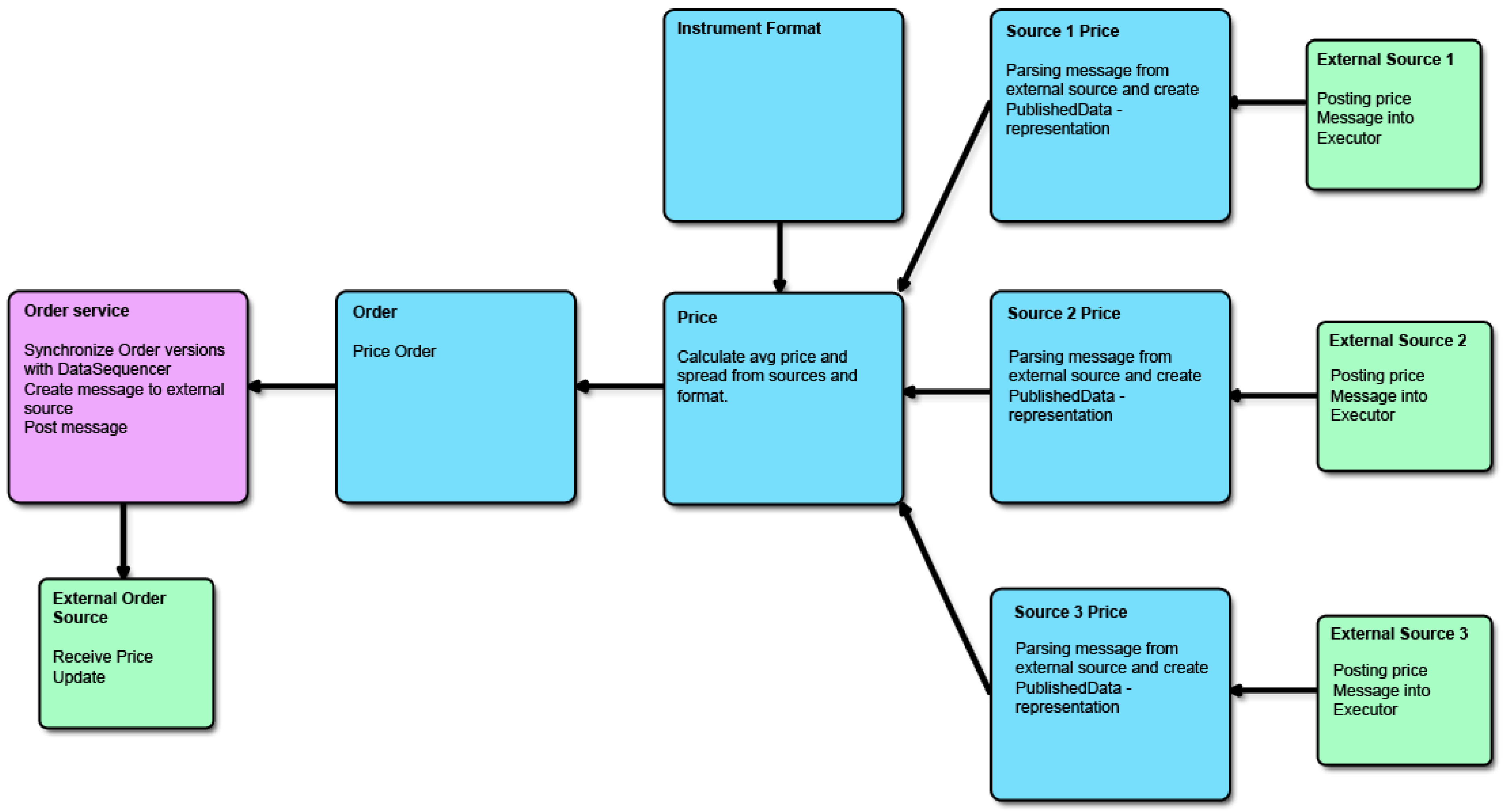
One of the problems when dealing with data that is updated concurrently by multiple threads is that things may happened at the same time. Even race conditions may occur where a newer version update beets its predecessor to deliver is content to its subscribers. In a lot of cases we are only interested in the latest state and may bypass this issue by always grabbing the entry from the STM as the update is triggered. If this is done within a transaction, we are ensured that we don't miss out on the last version.

On occasions where we cannot afford to miss a beat however (places where we use delta values or redistributes updates to external systems) A DataSequencer may be used. This object lies between the subscriber and the subscription. The sequencer only passes on the next version and holds premature updates until they are ready to be processed.

Performance

To benchmark performance I have created a fairly simple pricing service. The diagram below describes the concept. Green boxes are external systems. Blue boxes are STM entries and purple boxes are part of the system but outside the STM. The idea of the implementation is to have events be processed through four STM entries which I would consider to be reasonable in a real world scenario. The processing being done in this example, would in reality not require more than two entries, but for the sake of investigating the overhead of the synchronize mechanisms needed for this type of multi threaded system it has been somewhat over-designed.

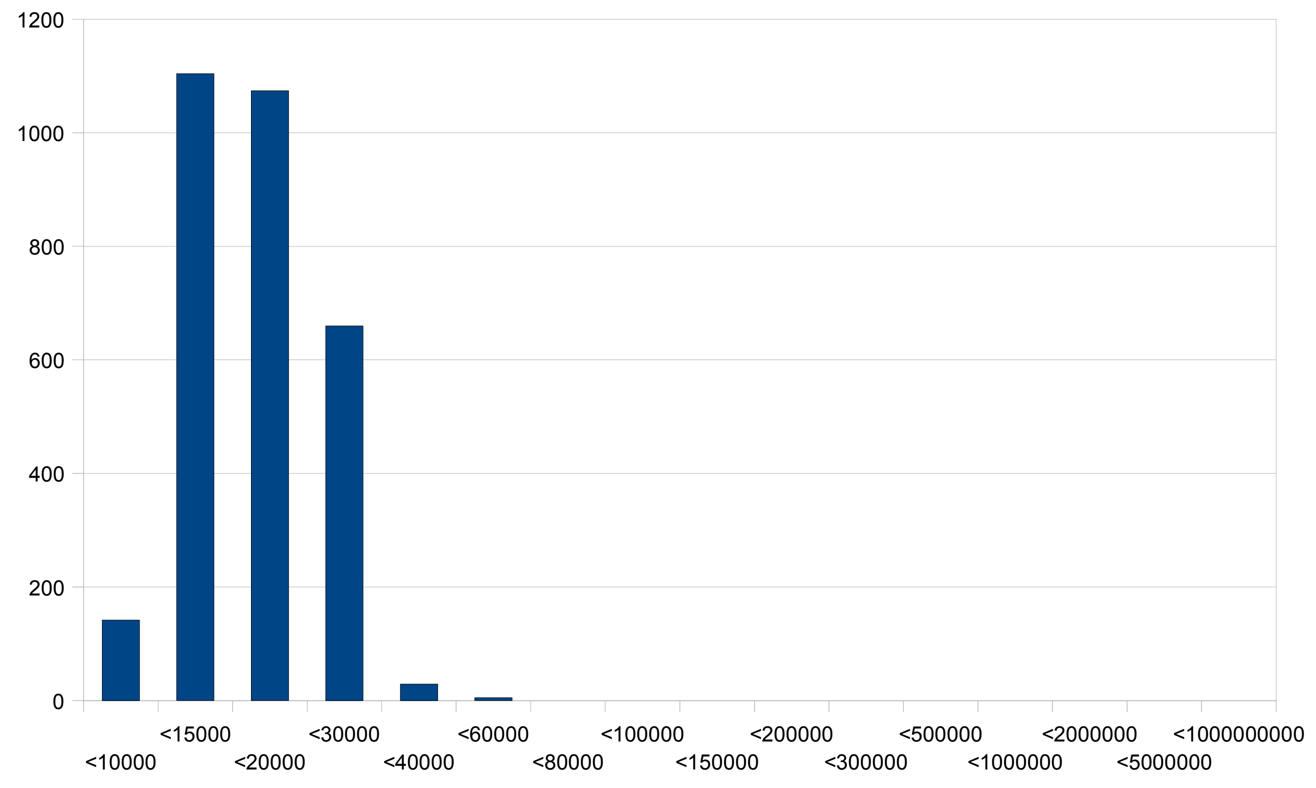
Requests are coming in to the system from the External Order source upon which the network of appropriate blue boxes is set up as STM entries and eventually sends a call to the External Source 1, 2, 3 to start delivering the requested price. These sources start posting their prices into the system executor and the 4 step processing is being done in the way the picture describes. Eventually the OrderObject is being sequences and posted back to the order service.


Instrument Format
Source 1 Price
Parsing message from external source and create PublishedData - representation
External Source 1
Posting price Message into Executor
Order service
Synchronize Order versions with DataSequencer
Create message to external source
Post message
Order
Price Order
Price
Calculate avg price and spread from sources and format.
Source 2 Price
Parsing message from external source and create PublishedData - representation
External Source 2
Posting price Message into Executor
External Order Source
Receive Price Update
Source 3 Price
Parsing message from external source and create PublishedData - representation
External Source 3
Posting price Message into Executor

I set the Order Server to request 1000 prices per second and listen to on average 20 updates before cancelling its requests.

The external price sources are updating their prices at a rate of 1000 time per second.

Y axis is number of events and X axis is the time in micro seconds it took to complete the event. The vast majority of the events are processed in under 20 microseconds.


1200
1000
800
600
400
200
0
<10000
<15000
<20000
<30000
<40000
<60000
<80000
<100000
<150000
<200000
<300000
<500000
<1000000
<2000000
<5000000
<1000000000