20th International Congress on Modelling and Simulation, Adelaide, Australia, 1–6 December 2013
www.mssanz.org.au/modsim2013

A distributed stream-processing infrastructure for computational models

F. Riedel [a] **and K. Watson** [a]

[a] *Fraunhofer Institute of Optronics, System Technologies and Image Exploitation (IOSB)*
Fraunhoferstr. 1, 76131, Karlsruhe, Germany
Email: felix.riedel@iosb.fraunhofer.de

Abstract: Decision support systems (DSS) that rely on time-sensitive information are demanding on the integration of computational models. Scientific models are commonly developed and tested with offline data coming from files and databases, but in a real-time DSS models have to deal with low-latency data streams, transmission faults and other imperfections. In practice, models need to process data from multiple data streams and various formats and require mechanisms to deal with delayed, missing and out-of-order data. It is desirable to handle data adaption, fault tolerance and other bookkeeping in a robust framework and allow domain experts to implement computational models in a mathematical language such as R, MATLAB or Fortran.

We present a platform that allows modellers to deploy R scripts and execute then in a distributed environment with online data. The platform is written in Java, dynamically sets up R sessions on distributed computers, manages the execution and deals with the input/output of models. An adapter strategy makes it possible to change data sources and formats without affecting the implementation of the computational model. In addition, fail-over mechanisms are implemented to guarantee processing in the face of a hardware or software fault.

In summary, the platform enables domain experts to implement concise computational models in a mathematical programming language (R), to test them offline in their accustomed environment and then to let them run online without modification in a fault-tolerant, distributed system. New models can therefore be easily added and the results are immediately usable by a real-time DSS.

Keywords: *real-time decision support, stream processing, online analysis, R integration, model execution, fault tolerance*

1 Introduction

As more and more data and data sources become available, not only the amounts of available data increases, but also the velocity of incoming data increases. Decision support systems (DSS) that rely on real-time data are expected to quickly update their information when new data becomes available. The path from raw data, to information, to knowledge should be as quick as possible. This requires models and systems which can provide continuous computations on low-latency, streaming data.

For scientists who develop models these requirements are challenging. Due to time constraints it is often not possible to reprocess all data that is used to compute a result. Therefore models have to be adapted in order to allow incremental updates or offline, multi-pass algorithms have to be replaced by online algorithms. But often enough this is not the primary obstacle preventing scientists from integrating new insights into data-driven DSSs. When models are implemented in mathematical languages, such as R, MATLAB or Fortran, there is lack of tools for deploying and maintaining algorithms in production systems. Just to wire up a model with streaming data sources which publish data asynchronously via a message-oriented middleware can be complex. Parsing proprietary message formats, handling delayed and out-of-order data records and joining multiple data streams are tasks that are usually easier to implement in a modern general-purpose programming languages such as Java. Consequently models that proved useful are either reimplemented or wrapped in a different programming languages. This approach is time-consuming, can introduce software bugs and slows down the process of integrating and testing new models.

In addition to the described integration gap, another complex issue is to is to deal with software, network and hardware faults. In a time-sensitive DSS the acceptable time for recovery is significantly lower than the time it takes to restart a model computation or a processing service. Therefore it is necessary to add resilience to the model execution.

Ideally the integration and deployment, and the fault-tolerant execution of models are handled by an appropriate framework. Unfortunately, to our knowledge there is no framework available satisfies these requirements, even though solutions for the individual issues exist.

In this paper we present an prototype system that combines multiple approaches into a framework that supports an effortless integration of R scripts into a message-driven continuous processing system and the implementation of simple fail-over mechanisms for incrementally updated computation processes.

2 Related Work

With the trend of moving from batch-processing to stream-processing several developments target the challenge of providing a resilient infrastructure for continuous computation and stream-processing. From the domain of Big Data technologies one of the most prominent projects is Storm [Marz, 2012], which calls itself a distributed real-time computation system. Storm allows developers to deploy and run so-called topologies which consist of so-called spouts and bolts, where spouts are stream sources and bolts are processing nodes or filters in terms of a pipes-and-filters architecture. The Storm runtime system distributes the processing over a cluster of worker nodes and guarantees that each record entering the processing topology is properly processed. If a worker node crashes or fails to process an item, the processing task is automatically reassigned to a different node. Storm employs an upstream backup [Hwang et al., 2005] approach in which upstream nodes keep their output until all depending operations completed successfully.

A similar system presented by Neumeyer et al. [2010] is S4, which stands for *Simple Scalable Streaming System*. It aims to be a general-purpose distributed stream computing platform. In comparison to Storm, S4 does not guarantee that each item is processed in the event of a failure or message loss. The allows S4 to avoid some overhead that is necessary for the implementation of such guarantees. Both, Storm and S4, run on the Java Virtual Machine (JVM) and are especially convenient to use with JVM languages. However, other languages can also be integrated by either writing a Java wrapper or, in the case of Storm, using Unix pipes and JSON data. There is no special support for R and the R environment has to be maintained separately by the user.

Another related group of systems are scientific workflow engines, such as Kepler [Altintas et al., 2004] and Taverna [Oinn et al., 2004]. Scientific workflow engines are data-flow oriented workflow engines aiming to provide convenient abstractions to create processing pipelines which integrate multiple tools and models into an overarching process. The main advantages of scientific workflow engines over simple hand-coded processes include ease-of-use, provenance tracking and parallelization. On the downside they were not developed stream processing applications and provide only insufficient features for fault-tolerant, distributed processing. The missing support for stream processing motivated [Neophytou et al., 2011] to develop CONFLuEnCE. CONFLuEnCE builds on top of Kepler and adds stream processing with windowing operations and push (message-driven) communication.

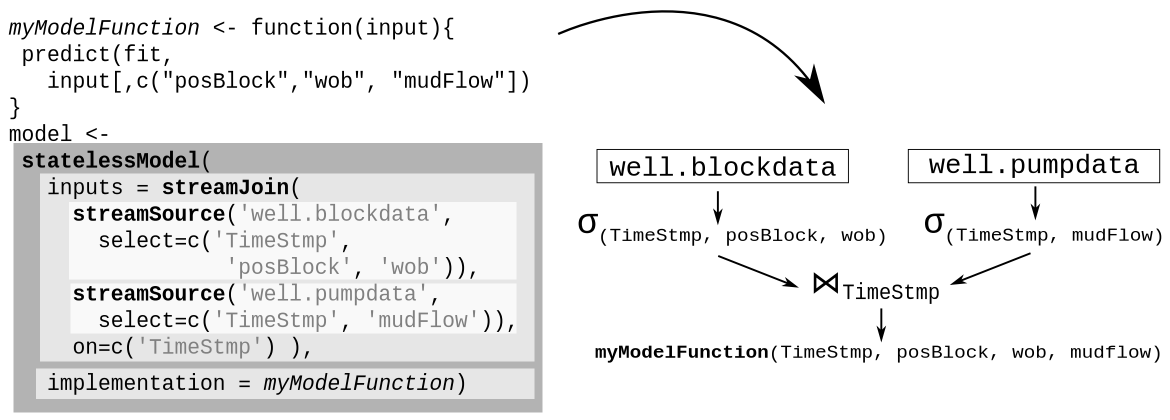
For the low-level integration of R scripts into production systems there has also been done a lot of ground work. There is of course the rJava [Urbanek, 2009] package which allows to use in principle any Java library from within R. The r-message-queue[1] package uses rJava to directly connect to messaging middleware. Another options is rzmq [Armstrong, 2011] which provides an R binding to the ZeroMQ messaging library. For integrating the R engine into a Java environment the Rserve package [Urbanek, 2003] and its Java client library can be used. Many more packages exist that could be mentioned here; however, they all have in common that they only provide low-level interfaces and leave a lot of work to the developer.

3 Deployment

In principle R scripts lend themselves for remote deployment, since the complete code of a model, including configuration data, can in general be serialized to single file via `save()` or the underlying functions. The serialized environment can then be easily transfered to a remote node and loaded into an R session. What is missing is a framework for the management of remote R installation, the deployment of new processes and the monitoring of active processes. However, this functionality is partly provided by Rserve [Urbanek, 2003], since Rserve provides a network interface that allows clients to spawn a remote-control R sessions. We will discuss later how we use Rserve in our system.

In addition to deployment and execution functionality we also need interfaces for getting input into and output from the R program. For R programmers it is most natural to load numerical vectors and data frames from files or databases, operate on these data structures and write the results again to files or databases. But in the world of stream processing we cannot or should not make the detour over persistent storage before processing the data. In our projects real-time data is usually distributed using a message-oriented middleware, which provides message delivery guarantees and a publish-subscribe mechanisms for communication. Therefore we assume that input data can be received by subscribing to right message topics and results should be published on a dedicated topic.

We propose a simple API that allows R programmers to describe the required input and let the framework deal with integration. Figure 1 shows a simplified example from one of our use cases. In the example we have a model that classifies sensor readings from a sub-surface drilling rig. During the development the scientist used a data frame as input that conveniently provided all sensor values for a given time stamp in a single row. In reality data about the block position and the drill bit are published on one topic (well.blockdata) and sensor readings from the mud circulation are published on another topic (well.pumpdata). As the source code on the left of figure 1 shows, our framework provides a convenient interface to select and join real-time data. As a result the programmer focus on the actual model.



```
myModelFunction <- function(input){
 predict(fit,
   input[,c("posBlock","wob", "mudFlow"])
}
model <-
 statelessModel(
   inputs = streamJoin(
     streamSource('well.blockdata',
       select=c('TimeStmp',
                'posBlock', 'wob')),
     streamSource('well.pumpdata',
       select=c('TimeStmp', 'mudFlow')),
     on=c('TimeStmp') ),
   implementation = myModelFunction)
```

well.blockdata
well.pumpdata
σ(TimeStmp, posBlock, wob)
σ(TimeStmp, mudFlow)
⋈TimeStmp
myModelFunction(TimeStmp, posBlock, wob, mudflow)

Figure 1: Specification of a stateless model in R

[1]https://code.google.com/p/r-message-queue/

The model description is read by a Java process which sets up the necessary subscriptions and input buffers. The Java process prepares the data and pushes it into an Rserve session which runs the actual R function. After the input row is processed the result is read by the Java process and published onto the message-oriented middleware. The output specification was omitted from this example, but the Java process can make use of various adapters that were implemented to encode and decode different data formats.

Figure 2 illustrates the general architecture of our implementation. We have several Rserve instances running on different machines. Each Rserve instance is managed by a Worker process which is implemented in Java. Each Worker process is a multi-threaded process which is in charge of managing Rserve sessions and handling the integration with the messaging middleware as described above. Whenever a new model needs to be deployed it is sent to the Supervisor which assigns the model to Workers based on available resources. The assignments, the model data and the resource usage information are shared over a Apache ZooKeeper cluster, which is also used for fault-detection and fail-over mechanisms as described in the next section.

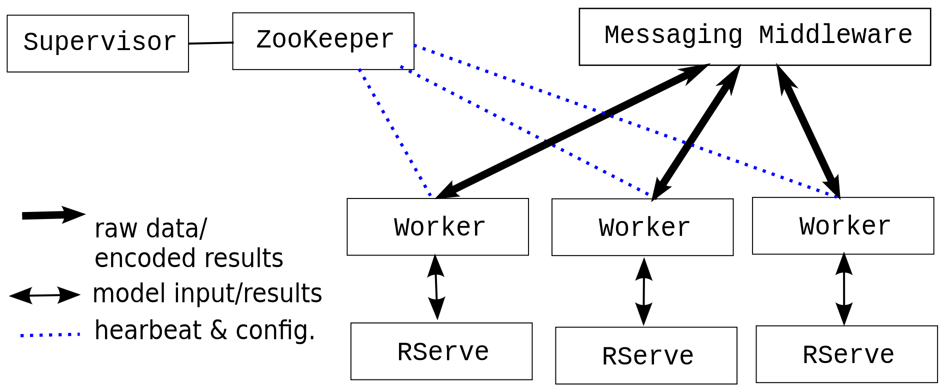


Figure 2: General Architecture

4 Fault-tolerance

In general a fault-tolerant system can be can be implemented using either a hot, cold or warm standby configuration. In a *hot standby* configuration an instance is mirrored by a second, redundant instance on a different machine. By having both instances running, any updates to the state are always automatically applied to both instances. In the event of an error condition an automatic switch-over (fail-over) guarantees availability without an interruption. The disadvantage of hot standby is of course the wastage of resources. This form of standby is also called *active standby* [Hwang et al., 2005].

In a cold standby configuration a second instance is only started after the failure of the primary. This requires the running instance to continuously persist its state to a reliable storage facility. Besides the requirement of a fast and reliable storage medium the main disadvantage is the possibly long recovery time. The model becomes available only after a new mode instance is created and the state is recovered from storage.

The third variant of standby, *warm standby*, is a compromise between hot and cold standby. As in hot standby a second instance is kept available, but it does keep its state always up-to-date; instead, a snapshot (checkpoint) of the primary's state is periodically loaded by the standby instance. This approach, also called *passive standby* [Hwang et al., 2005], saves resources as the primary instance does not have to persists its state on every update and the secondary does not have to compute all updates. The disadvantage is of course, that in case of a failure the standby works off an out-dated checkpoint and the latest state updates are lost. This problem can be avoided by the *upstream backup* approach which keeps inputs in the upstream's output queue up to the last checkpoint. Any inputs between the checkpoint and the recovery can therefore be replayed to the new instance.

In general, hot standby is the best option when there are no relaxations on the latency or the accuracy requirements. We therefore chose to implement a hot standby configuration in our system. However, as we will see later cold and warm standby can easily be added using the same infrastructure.

Even though the hot standby configuration does require to save or load the state during normal operations, after a failure a new hot standby instances need to be instantiated whose state has to be brought into sync with the primary's state. There are different ways to achieve this, depending on the type of model we deal with. In our system we differentiate between three types of models:

- **stateless** The results of a stateless model depends only on the current input data and configuration data (e.g. model parameters), which does not depend on any previous executions of the model.

  Examples: point based classifiers, thresholds

- **sliding-window state** This group consists of models that implicitly or explicitly keep a size or time limited window of past input data. Results from these models depend on this window of previous observations in addition to current input data and configuration data.

  Examples: activity and temporal pattern recognition, moving average filter

- **stateful** The results of fully stateful models depend on all previous input data in addition to the current data and configuration data.

  Examples: Kalman filter, heavy hitter detection, online learner

Stateless models are the easiest to deal with. They have no state and a new instance therefore does not need any state information to become a valid hot standby.

For models with a sliding-window state we have two options. The first option applies if the length of the sliding-window significantly smaller than the mean time between failures. In this case we can choose to be lazy and wait until the window of the new instance fills up with new input data. This lazy approach causes the system to be without a standby for the duration of a time window, but also saves us from any extra work. Depending on the length of the window this is a good trade-off. The alternative is to treat models with sliding-window states the same as general stateful models.

In the case of stateful models we need to mirror the state of the active instance. Checkpointing and loading the checkpoint on the standby instance takes in general longer than the time between the arrival of new input data. Therefore, the new instance buffers inputs that are published during the checkpointing operation. After the checkpoint is loaded the buffered input is replayed to model in order to bring the model up to speed.

4.1 Implementation

The implementation of hot standby and fail-over requires reliable mechanisms for fault detection and a distributed leader election protocol to select active model among possibly multiple redundant instances. The problem of leader election is a special case of the more general problem of reaching consensus among distributed processes. The Paxos algorithm [Chandra et al., 2007] offers a solution to this problem and with Apache ZooKeeper [Hunt et al., 2010] we have an implementation that provides an API that can be used to implement solutions to various coordination problems – including leader election. As ZooKeeper already uses heartbeats to detect client failure, we do not need an additional heartbeat mechanism.

Currently whenever a new model is deployed two Workers are assigned to run the model. Both workers immediately enter into a leader election and only the primary instance (leader) will publish its model's results onto the messaging middleware. The timestamp of the last processed record is written to a shared variable in ZooKeeper. When the primary instance fails the hot standby gets elected and starts publishing its results. The stored timestamp is used to determine where the failed instance left off.

All this logic is contained within the Java Worker implementation. If the Rserve process or an R session fails, the managing Worker forfeits all affected leaderships and disconnects corresponding ZooKeeper sessions to declare itself failing. When Rserve can be restarted or only a session was lost the Supervisor will again start assigning tasks to the node.

As we discussed above the actual work for fault-tolerance framework is in creating instances that mirror the state of the primary instance. For this work we need some additional information. Figure 3 shows the definition of a sliding-window model and a stateful model. For the sliding-window model it is sufficient to specify the size of the window. It will always get a data frame as input with number of rows matching the window size. The sliding window is implemented in Java as a circular buffer. When the state needs to be replicated the Java framework can either follow the lazy approach with delayed instantiation or us R-independent functions to

```
myModelFunctionB <- function(input){...}

modelB <-
 windowModel(
   inputs = streamSource('example'),
   output = 'window.output',
   implementation = myModelFunctionB,
   windowsize = 100
 )
```

```
serializeModel <- function(){...}
deserializeModel <- function(){...}
myModelFunctionC <- function(input){...}

modelC <-
 statefulModel(
   inputs = streamSource('example'),
   output = 'stateful.output',
   implementation = myModelFunctionC,
   serializer = serializeModel,
   deserializer = deserializeModel
 )
```

Figure 3: Specification of a sliding-window model and a stateful model

serialize and deserialize the window of data. Both is transparent to the R code. So the actual R model becomes in fact stateless; the state is kept only in the Java Worker.

For general stateful models this is not possible since the Worker does know the state of the model; hence, it is necessary to provide a function to serialize and one deserialize the state of a model (cp. figure 3). When the state needs to be replicated to new instance, the Worker of the new instance starts buffering inputs, then the primary Worker pauses execution of model while buffering inputs and calls the serialization function to get a checkpoint of the state. When the checkpointing is finished the primary instance resumes work while the new standby instance reads in the checkpoint and replays the inputs since the checkpoint.

5 Conclusion and Future Work

We presented a framework that can be used to easily deploy and integrate R models into message-driven environment in order to provide continuous computations to time-sensitive decision support systems. Unlike existing frameworks our prototype combines ease-of-use and convenient R integration with reliable fail-over mechanisms. We expect that our development leads to shorter development cycle and considerable time savings.

While the current prototype is already usable, during our work we identified several issues that can be improved. For the deployment it is necessary that all required packages are installed on the worker nodes. Currently this is done by automatically installing the most recent versions of all required packages. This can lead to problems if packages are not backward compatible. Therefore it is necessary to be able to specify a package version requirement. While this is a general problem in the R package management, the issue is more problematic when multiple models share an R installation that they have no or little control over.

The chosen path of implementing fault-tolerance consciously prioritizes recovery time over resource usage. When the system is used in a larger scale priorities are likely to shift and it becomes necessary to use more sophisticated, resource-saving models of fault-tolerance. We have also not yet evaluated where the limits for ZooKeeper are. In general ZooKeeper is not optimized for writes and should not be used for larger amounts of data. When we deal with a large amount of processes and models with large state data, both might become a problem.

Acknowledgement

The research leading to these results was co-funded by the European Commission under FP7 (Seventh Framework Programme) Action Line: ICT-2009.4.3 *Intelligent Information Management* Project Reference: 258723

References

Altintas, I., C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock (2004). Kepler: an extensible system for design and execution of scientific workflows. *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.*, 423–424.

Armstrong, W. (2011). rzmq: R binding for zmq (http://www.zeromq.org/).

Chandra, T., R. Griesemer, and J. Redstone (2007). Paxos Made Live - An Engineering Perspective. In *PODC '07: 26th ACM Symposium on Principles of Distributed Computing.*

Hunt, P., M. Konar, F. Junqueira, and B. Reed (2010). ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference*.

Hwang, J.-H., M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik (2005). High-Availability Algorithms for Distributed Stream Processing. *21st International Conference on Data Engineering (ICDE'05)*, 779–790.

Marz, N. (2012). Storm: Distributed and fault-tolerant realtime computation. In *Open Source Conference (OSCON)*. O'Reilly.

Neophytou, P., P. Chrysanthis, and A. Labrinidis (2011). CONFLuEnCE: Implementation and application design. *Proceedings of the 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing*.

Neumeyer, L., B. Robbins, A. Nair, and A. Kesari (2010, December). S4: Distributed Stream Computing Platform. *2010 IEEE International Conference on Data Mining Workshops*, 170–177.

Oinn, T., M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li (2004, November). Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics (Oxford, England) 20*(17), 3045–54.

Urbanek, S. (2003). Rserve – A Fast Way to Provide R Functionality to Applications. In K. Hornik, F. Leisch, and A. Zeileis (Eds.), *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, Number Dsc, Vienna, Austria.

Urbanek, S. (2009). *rJava: Low-Level R to Java Interface*.