Generalized Synchronizations and Capacity Constraints for Java Modelling Tools

Giuliano Casale
Imperial College London, UK
g.casale@imperial.ac.uk

Mattia Cazzoli
Politecnico di Milano, Italy
mattia.cazzoli@mail.polimi.it

Shuai Jiang
Imperial College London, UK
shuai.jiang14@imperial.ac.uk

Vitor S. Lopes
Imperial College London, UK
v.soares-lopes@imperial.ac.uk

Giuseppe Serazzi
Politecnico di Milano, Italy
giuseppe.serazzi@polimi.it

Lulai Zhu*
Imperial College London, UK
lulai.zhu15@imperial.ac.uk

ABSTRACT

Java Modelling Tools (JMT) is a suite of performance evaluation tools based on queueing network models. Recently *JSIMgraph*, the JMT discrete-event simulation tool, has been extended to express features of current computing systems such as Big data applications. The goal of this demonstration is to showcase novel support in JMT for fork-join synchronization, dynamic scaling of parallelism levels, memory and group capacity constraints.

Keywords

Queueing network, simulation, synchronization, JMT

1. INTRODUCTION

Java Modelling Tools (JMT) is an integrated framework of tools for performance evaluation of computer systems based on queueing network models [1]. This features primarily discrete-event simulation and mean value analysis algorithms, among others. In the most recent releases (JMT 0.9.4 and 1.0.0) we have substantially extended the expressiveness of the simulation models in *JSIMgraph*, JMT's queueing network simulator, in order to simplify the modelling of complex systems with advanced synchronization types and more flexible capacity constraints. While still classifiable as queueing networks, the resulting models extend the types of nodes that are typically considered in queueing network theory. Moreover, certain components that facilitate the modelling of specific technologies such as Hadoop/MapReduce have been added to JMT.

1.1 Advanced Synchronization Strategies

JSIMgraph has supported since several years the definition of fork-join subsystems. Two simulation components, the *Fork* and the *Join* nodes, disassemble a job into tasks then re-assemble them in the original job, respectively. More precisely, once a job enters a Fork, it generates an identical user-customizable number of tasks to each output link. Later on, the Join node waits for all such tasks to arrive in order to merge them back in the original job. An extension of this mechanism was provided with two main goals: i) allow the customization of the number of tasks sent on each individual link outgoing from the Fork node; ii) allow flexibility in the synchronization policy adopted at the Join node.

1.1.1 Advanced Fork Strategies

The new *Branch Probabilities Fork Strategy* enables the user to specify the probability that the k-th outgoing link from the Fork node will receive one or more tasks. For each link, the user may also specify the distribution of the number of tasks sent on that link for each forked job. A different set of probabilities can be specified for each job class. Such fine-grained settings allows to precisely control the generation of synchronized batches going from a source (the fork) to processing systems.

The *Random Subset Fork Strategy* allows users to specify the probability that the Fork sends tasks to k outgoing branches, which will be then chosen randomly among the n available output branches of the Fork. This abstraction may be useful, for instance, to represent systems where work is replicated across a subset of servers to mitigate the risk of contention.

Two additional fork strategies, called *Class Switch* and *Multi-Branch Class Switch* strategies, allow *JSIMgraph* to instantaneously change the class of all the sibling tasks, differentiating it from the class of the job from which they were forked. In the Multi-Branch case, a different mix of classes can be generated on each outgoing link of the Fork.

1.1.2 Advanced Join Strategies

The standard join strategy waits for all the forked tasks to wait at the same join node prior to merge them all and output the original job. Novel variants of this policy are the *Quorum* strategy, which allows Join nodes to wait only k out of n forked tasks prior to re-assembling the original job, ignoring the subsequent arrivals of other tasks from the same job, and the *Guard* policy which further refines this notion by allowing to wait for a specific mix of jobs, where a mix is a given combination of job classes.

1.1.3 Semaphore Synchronization

The *Semaphore* is a new *JSIMgraph* node inspired by the *slow start* mechanism present in Hadoop/MapReduce (HMR). In HMR a job is sequentially processed in map and reduce phases. A number of key-value pairs are produced at the end of the map phase, then

*The authors thank Phumin Phuangjaisri and Marco Gribaudo for contributions and helpful discussions. This research has been partially funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644869 (DICE).

ICPE'17 April 22-26, 2017, L'Aquila, Italy

ACM ISBN 978-1-4503-4404-3/17/04.

DOI: http://dx.doi.org/10.1145/3030207.3053666

sorted and shuffled, and finally delivered to the reducers. With slow start, the reducers can reduce network contention by starting to collect some of the produced key-value pairs before the map phase is concluded. This mechanism is triggered when a certain *percentage* of the total number of mappers has completed.

Assuming to represent key-value pairs as forked tasks of an high-level HMR job, one may attempt to model slow start using a Quorum strategy, where k out of n key-value pairs are transferred to the reduce phase. This is however incorrect since the Quorum strategy would ignore the remaining $n - k$ tasks, removing them from the simulation once they reach the Join node. Thus the first k tasks would be merged into a job circulating *outside* the Fork-Join section sent to the reducers, whereas $n - k$ valid tasks are still cycling *inside* the section while finishing service at the mappers.

The Semaphore instead provides a node that temporarily *blocks* a set of tasks inside a Fork-Join subsystem. For each forked job the Semaphore will wait for exactly k tasks. Once they have all entered the Semaphore, they will be released *as tasks*, i.e., without any Join. Any subsequent task forked from the the same job that arrives to the Semaphore is not delayed.

1.1.4 Scaler

The *Scaler* is a new *JSIMgraph* node motivated by the problem of modelling the Directed Acyclic Graph (DAG) used to model Apache Spark jobs. Spark jobs comprise a series of operators on data, arranged according to a DAG. DAGs are split into map and reduce stages that are then scheduled for execution.

The DAG structure of Spark jobs can be captured by JMT: a network of queues normally represents the processing elements, whereas the degree of parallelism in each stage and the synchronisation between stages may be controlled using the advanced Fork-Join policies. However, it is foreseeable that large DAGs where stages see frequent changes in the parallelism level of the application may become cumbersome to model.

The Scaler addresses this design issue by providing in a single element a Join followed by a Fork. Hence, the Scaler receives k tasks and outputs n tasks. The advantage of the Scaler is to offer in a single element the ability to change parallelism level from k to n, instead of having to include separate Join and Fork elements, which would complicate the graphical representation of the model and slow down the simulation.

1.2 Advanced Capacity Constraints

A finite capacity region (FCR) is a subnetwork with constraints on the number of jobs that can be inside it at any given time. Such limits can refer to the total number of jobs in the network, or to the jobs of specific classes, or both. This has been a standard element of JMT since the early releases, which can be used to model features such as resource pooling and software limits to thread parallelism. However, in practice it is often the case that also memory is a limiting factor in deciding whether to schedule a thread by a software system or not. Moreover, constraints may exist only to prevent only certain specific mixes of classes to run altogether. The latest version of JMT supports these extensions.

1.2.1 Memory Capacity Constraints

A memory constraint associates to each class a *memory weight*. Each FCR has a total *memory capacity* M which is temporarily depleted by jobs that enter the FCR. If the kth job inside the FCR carries a memory weight m_k, then the available memory is $M - \sum_k m_k$. A new job is admitted to the FCR only if it satisfies all finite capacity constraints and its memory requirement fits in the available memory. Note that the notion of memory is here entirely deterministic, as JMT does not support yet the definition of a time-varying memory weight. Clearly, this new feature can help in modelling the memory behavior of software systems. A new metric, called *FCR Memory Weight*, provides the current memory value of $\sum_k m_k$, i.e., the currently allocated memory inside the FCR.

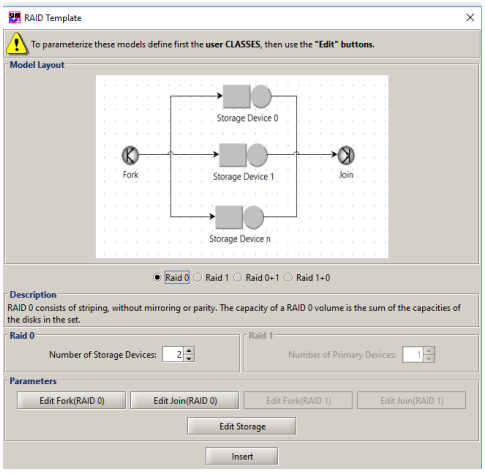

Figure 1: JSIMgraph template: RAID disk modelling

1.2.2 Group Capacity Constraints

As mentioned, a FCR constraints the total and per-class number of jobs inside the region. However, it is possible that in some cases the user wants to constraint certain groups of jobs belgoning to a subset of classes. For example, one could set that if a group is formed by two classes A and B and a capacity constraint is set to 5 jobs, then the sum of A and B jobs inside the FCR must not exceed 5 at any time. This feature is helpful in particular for cases where a single class A is split into multiple classes $A_0, A_1, \ldots$ by means of a *ClassSwitch* simulation element. By listing the said classes in a group, a constraint can be put in place effectively limiting the parallelism of class A without having to change all classes A_i back into A, prior to entering the FCR.

1.3 Templates

Templates are a new feature for *JSIMgraph*. They are downloadable modules used to automate the sequence of operations required to create a complex simulation model for a specific technology. They can be dynamically retrieved either from the official server from servers run by third parties who want to enrich the JMT functionalities. Currently, the offered templates range from creation of three-tier application models, to modelling of RAIDs and the CEPH storage system. A user interface allows to browse a tree of the available templates, as shown in Figure 1.

2. REFERENCES

[1] M. Bertoli, G. Casale, and G. Serazzi. Java modelling tools: an open source suite for queueing network modelling and workload analysis. In *Proc. of QEST*, 119–120. IEEE, 2006.

[2] M. Bertoli, G. Casale, and G. Serazzi. The jmt simulator for performance evaluation of non-product-form queueing networks. In *Proc. of the 40th Annual Simulation Symposium (ANSS)*, 3–10, 2007.