Simulator of "Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems"

Xiaoxing Ma[1,2], Luciano Baresi[1], Carlo Ghezzi[1], Valerio Panzica La Manna[1], and Jian Lu[2]

[1]*Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University, China*
{*ma,baresi,ghezzi,panzica*} *@elet.polimi.it, lj@nju.edu.cn*

June 7, 2011

1 How to download and compile the simulator

To download and run the simulator, one should:

- Be sure JDK 1.6, Eclipse and Subclipse (Eclipse SVN plugin) are properly installed
- Start Eclipse and switch to $SVNRepositoryExploring$ perspective
- Add a new location with the following URL: `http://vcsim.googlecode.com/svn`
- Check out folder `public` as a project in the workspace (project name: `VCSim2`).
- Switch to the $Java$ perspective
- If needed, include all jar files in directory lib in the buildpath/classpath

One should read the paper in directory `doc` to understand the aim of the simulation. S/he should also set the options of the JVM as follows: `-Xms1000m -Xmx1000m` to allocate enough heap memory for the simulator.

2 Timeliness and Disruption

To evaluate the timeliness and disruption of our approach with respect to systems of different sizes and different network latencies, one must configure program `TimelinessDisruptionTargetRandom.java` (in package `it.polimi.vcdu.exp.run`) with the number of components Cnm and network delay Dly for the experiment. Different runs with the same number of components, but with different delays, allow one to evaluate the impact of network latency; the same latency and varying sizes help study the impact of the system's size. Each run creates a file `newexp_V[Cnm]E2D[Dly]N100TargetRandomServerNodes.csv`, under directory `resultsExperiments/timelinessDisruption/`, to store produced results.

Each execution comprises 7 steps (see method `ExperimentRecordReplay.run()`):

1. `expRecord` records the randomly-generated scenario, including the injection of root transactions (both when and to which component it is injected) and the progress of each transaction (when it initiates its sub-transactions and on which neighbor components).
2. `expQuiescence` runs the recorded scenario by using the quiescence approach. Note that `ReqTime` is the time instance at which a reconfiguration request is received, `workRequestMQFC` states the work done at this time, `quiescenceTime` is the time instant at which the target component is quiescent and `workQuiescenceQ` says the work done at this time.

3. `expOnDemandVersConsistency_Blocking` runs the recorded scenario by using the on-demand version consistency with the blocking strategy for freeness. Note that `ReqTime` is the time instance at which a reconfiguration request is received, `workRequestMQFC` states the work done at this time (it should be the same as in step 2 above because the set-up is on-demand), `vcFreenessTime` is the time instant at which the target component is quiescent and `workFreenessF` says the work done at this time.

4. `expOnDemandVersConsistency_ConcurrentVersions` runs the recorded scenario by using the on-demand version consistency with the concurrent versions strategy for freeness. Note that `ReqTime` is the time instance at which a reconfiguration request is received, `workRequestMQFC` states the work done at this time (it should be the same as in step 2 above because the set-up is on-demand), `concurVersTime` is the time instant at which the target component is free and `workConcurVersFreenessC` says the work done at this time.

5. `expMeasuringQuiescence` runs the recorded scenario and does not update the component. It measures `workQuiescenceM`, that is, the amount of work it would have done at `quiescenceTime` if there had been no dynamic updates.

6. `expMeasuringODVC_Blocking` runs the recorded scenario and does not update the component. It measures `workFreenessM`, that is, the amount of work it would have done at `vcFrenessTime` if there had been no dynamic updates.

7. `expMeasuringODVC_ConcurrentVersions` runs the recorded scenario and does not update the component. It measures `workConcurVersFreenessM`, that is, the amount of work it would have done at `concurVersTime` if there had been no dynamic updates.

The simulator is configured to run each experiment 100 times. With these values we can compute the timeliness of the different approaches, that is, quiescence, version consistency with the blocking strategy, and version consistency with concurrent versions:

$$delta_{quiescence} = quiescenceTime - ReqTime \tag{1}$$

$$delta_{blocking-strategy} = vcFreenessTime - ReqTime \tag{2}$$

$$delta_{concurrent-versions} = concurVersTime - ReqTime \tag{3}$$

and their corresponding disruptions:

$$lostWork_{quiescence} = workQuiescenceM - workQuiescenceQ \tag{4}$$

$$lostWork_{blocking-strategy} = workFreenessM - workFreenessF \tag{5}$$

$$lostWork_{concurrent-versions} = workConcurVersFreenessM - workConcurVersFreenessC \tag{6}$$

3 Freeness

To compare the BF strategy to freeness against the WF strategy under different workloads of the system, one must run `WaitingBlocking.java` (in package `it.polimi.vcdu.exp.run`). S/he can vary the mean arrival intervals of root transactions: suggested values are between 1600 and 250 for a 16-node system with mean local processing time set to 50. The program prints produced results directly on the console.