Configuring Simulations in OMNeT++ 4.0

Changes in Ini File Syntax and Features

András Varga (andras @ omnetpp.org)

July 9, 2007

1 Introduction

In OMNeT++ 4.0, we re-thought and improved the concept of ini files and configuring simulations. Major enhancements include the introduction of named configurations which can build on one another, and the ability to define various scenarios by iterating over a parameter space, for batch execution. As you will see, the introduction of scenarios has created a clean way to do multiple runs with the same parameters but different random number seeds, and with scenario meta-information saved into the result files one will get a much more organized view in the result analysis phase about the simulations performed and their results.

With the above goals in mind, configuration reading and handling was re-implemented from scratch. Additional benefits from this re-implementation are greatly improved performance (we use maps instead of linear searches), the ability to catch and report mistyped configuration keys in the ini files, and to be able to print a list of all supported configuration keys, complete with descriptions.

OMNeT++ 4.0 ini files have roughly the same syntax as 3.x ini files, but the design goals could not be achieved without rationalizing some aspects of the ini files, like changing the names of some sections and config keys. This breaks backward compatibility, and thus OMNeT++ 3.x ini files need to be converted to be used with OMNeT++ 4.0. There is a command-line tool to do this conversion, or you can leave it to the IDE's Inifile Editor, or even doing it manually is quite painless and straightforward.

The remaining sections describe the changes and enhancements of ini file handling in OMNeT++ 4.0.

2 Sections

2.1Overview

To begin with, the [Cmdenv], [Tkenv], [Parameters], [OutVectors] and [Partitioning] sections are no more -- only [General] is left. The organizing principle of sections is no longer the duality of topics (Cmdenv, Tkenv, output vectors, etc.) and runs (run 1, run 2, etc.), but named configurations -- only, and always.

Named configurations are sections of the form [Config <configname>], where <configname> is by convention a camel-case string that starts with a capital letter: "Config1", "WirelessPing", "OverloadedFifo", etc. For example, omnetpp.ini for an Aloha simulation might have the following skeleton:

```
[General]
...
[Config PureAloha]
```

```
...
[Config SlottedAloha1]
...
[Config SlottedAloha2]
...
```

Keys formerly in the [Cmdenv] section simply got a "cmdenv-" prefix, and were moved to the [General] section. Similarly, [Tkenv] keys got a "tkenv-" prefix, and were moved to [General]. The contents of the rest of the disappearing sections ([Parameters], [OutVectors] and [Partitioning]) just got merged into [General]. This way, ini files actually became simpler and less error-prone: it cannot a happen any more that a configuration key gets ignored just because it was mistakenly added to the wrong section.

> **NOTE**: A few, less common configuration keys have changed too, like **.interval= for specifying the output vector data collection interval has been renamed to **.recording-interval=, so please check the section on migrating ini files before setting out to manually convert files, and/or print the list of supported configuration keys by invoking any simulation executable with the "-q configdetails" command-line option.

There are also no [Run X] sections any more: when converting a 3.x ini file, change them to named configurations. Automatic conversion tools will just generate names like [Config One], [Config Two], etc, and it is up to you to change them to better names afterwards.

> **NOTE**: Another change the conversion script does is replace the semicolons used as comment marks with hash marks ("#"). Semicolon is no longer a comment mark in OMNeT++ 4.0 inifiles.

Some configuration keys (such as user interface selection) are only accepted in the [General] section, but most of them can go into Config sections as well.

The description= configuration key still exists (you can provide a string there, with an explanation about the purpose or content of the section), but it has less significance now that configurations can be given descriptive names instead of just being called [Run 1], [Run 2], etc.

When you run a simulation, you need to select one of the configurations to be activated. In Cmdenv, this is done with the '-c' command-line option:

```
% aloha -c PureAloha
```

The simulation will then use the contents of the [Config PureAloha] section to set up the simulation. (Tkenv, of course, lets you select the configuration from a dialog.)

2.2Fallbacks

Actually, when you activate the PureAloha configuration, the contents of the [General] section will also be taken into account: if some configuration key or parameter value is not found in [Config PureAloha], then the search will continue in the [General] section. In other words, lookups in [Config PureAloha] will fall back to [General]. The [General] section itself is optional; when it is absent, it is treated like an empty [General] section.

All named configurations fall back to [General] by default. However, for each configuration it is possible to specify a fall-back section explicitly, using the extends= key. Consider the following ini file skeleton:

```
[General]
...
[Config SlottedAlohaBase]
...
[Config SlottedAloha1]
extends = SlottedAlohaBase
...
[Config SlottedAloha2]
extends = SlottedAlohaBase
```

```
...
[Config SlottedAloha2a]
extends = SlottedAloha2
...
[Config SlottedAloha2b]
extends = SlottedAloha2
...
```

If you activate the SlottedAloha2b configuration, lookups will consider sections in the following order (this is also called the "section fallback chain"): SlottedAloha2b, SlottedAloha2, SlottedAlohaBase, General.

The effect is the same as if the contents of the sections SlottedAloha2b, SlottedAloha2, SlottedAlohaBase and General were copied together into one section, one after another, [Config SlottedAloha2b] being at the top, and [General] at the bottom. Lookups always start at the top, and stop at the first matching entry.

The concept is similar to inheritance in object-oriented languages, and benefits are similar too: you can to factor out the common parts of several configurations into a "base" configuration, and the other way round, you can reuse existing configurations (as opposed to copying them) by using them as a base. In practice you will often have "abstract" configurations too (in the C++/Java sense), which assign only a subset of parameters and leave the others open, to be assigned in derived configurations.

If you are experimenting a lot with different parameter settings of a simulation model, these techniques will make it a lot easier to manage ini files.

3 Scenarios

3.1Basic use

It is quite common in simulation studies that the simulation model is run several times with different parameter settings, and the results are analyzed in relation to the input parameters. OMNeT++ 3.x had no direct support for batch runs, and users had to resort to writing shell (or Python, Ruby, etc.) scripts that iterated over the required parameter space, and generated a (partial) ini file and run the simulation program in each iteration.

OMNeT++ 4.0 largely automates this process, and eliminates the need for writing batch execution scripts. It is the ini file where the user can specify iterations over various parameter settings. Here's an example:

```
[Config AlohaScenario]
*.numHosts = ${1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential( ${0.2, 0.4, 0.6} )
```

This scenario expands to 8*3 = 24 simulation runs, where the number of hosts iterates over the numbers 1, 2, 5, 10, 20, 30, 40, 50, and for each host count three simulation runs will be done, with the generation interval being exponential(0.2), exponential(0.4), and exponential(0.6).

How does it get run? First of all, Cmdenv with the '-n' option will tell you how many simulation runs a given section expands to. (You'll of course use Cmdenv for batch runs, not Tkenv.)

```
% aloha -u Cmdenv -c AlohaScenario –n

OMNeT++/OMNEST Discrete Event Simulation
...
Config: AlohaScenario
Number of runs: 24
```

If you add the '-g' option, the program will also print out the values of the iteration variables for each run. Note that the scenario description actually maps to nested loops, with the last "${..}" becoming the innermost loop. The iteration variables are just named $0 and $1 -- we'll see that it is possible to give meaningful names to them. Please ignore the '$repetition=0' part in the printout for now.

```
% aloha -u Cmdenv -c AlohaScenario -n -g
OMNeT++/OMNEST Discrete Event Simulation
...
Config: AlohaScenario
Number of runs: 24
Run 0: $0=1, $1=0.2, $repetition=0
Run 1: $0=1, $1=0.4, $repetition=0
Run 2: $0=1, $1=0.6, $repetition=0
Run 3: $0=2, $1=0.2, $repetition=0
Run 4: $0=2, $1=0.4, $repetition=0
Run 5: $0=2, $1=0.6, $repetition=0
Run 6: $0=5, $1=0.2, $repetition=0
Run 7: $0=5, $1=0.4, $repetition=0
...
Run 19: $0=40, $1=0.4, $repetition=0
Run 20: $0=40, $1=0.6, $repetition=0
Run 21: $0=50, $1=0.2, $repetition=0
Run 22: $0=50, $1=0.4, $repetition=0
Run 23: $0=50, $1=0.6, $repetition=0
```

Any of these runs can be executed by passing the '-r <runnumber>' option to Cmdenv. So, the task is now to run the simulation program 24 times, with '-r' running from 0 through 23:

```
% aloha -u Cmdenv -c AlohaScenario -r 0
% aloha -u Cmdenv -c AlohaScenario -r 1
% aloha -u Cmdenv -c AlohaScenario -r 2
...
% aloha -u Cmdenv -c AlohaScenario -r 23
```

This batch can be executed either from the OMNeT++ IDE (where you are prompted to pick an executable and an ini file, choose the scenario from a list, and just click Run), or using a little command-line batch execution tool supplied with OMNeT++.

Actually, it is also possible to get Cmdenv execute all runs in one go, by simply omitting the '-r' option.

```
% aloha -u Cmdenv -c AlohaScenario

OMNeT++/OMNEST Discrete Event Simulation
Preparing for running configuration AlohaScenario, run #0...
...
Preparing for running configuration AlohaScenario, run #1...
...
...
Preparing for running configuration AlohaScenario, run #23...
```

However, this approach is not recommended, because it is more susceptible to C++ programming errors in the model. (For example, if any of the runs crashes, the whole batch is terminated -- which may not be what the user wants).

Let us get back to the ini file. We had:

```
[Config AlohaScenario]
*.numHosts = ${1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential( ${0.2, 0.4, 0.6} )
```

The ${...} syntax specifies an iteration. It is sort of a macro: at each run, the whole ${...} string gets textually replaced with the current iteration value. The values to iterate over do not need to be numbers (unless you want to use the *"a..b"* or *"a..b step c"* syntax), and the substitution takes place even inside string constants. So, the following examples are all valid (note that textual substitution is used):

```
*.param = 1 + ${1e-6, 1/3, sin(0.5)}
    ==> *.param = 1 + 1e-6
        *.param = 1 + 1/3
        *.param = 1 + sin(0.5)

*.greeting = "We will simulate ${1,2,5} host(s)."
    ==> *.greeting = "We will simulate 1 host(s)."
        *.greeting = "We will simulate 2 host(s)."
        *.greeting = "We will simulate 5 host(s)."
```

To write a literal ${..} inside a string constant, quote "{" with a backslash, and write "$\{..}".

3.2Named iteration variables

You can assign names to iteration variables, which has the advantage that you'll see meaningful names instead of $0 and $1 in the Cmdenv output, and also lets you refer to the variables at more than one place in the ini file. The syntax is ${<varname>=<iteration>}, and variables can be referred to simply as ${<varname>}:

```
[Config Aloha]
*.numHosts = ${N=1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential( ${mean=0.2, 0.4, 0.6} )
**.greeting = "There are ${N} hosts"
```

The scope of the variable name is the section that defines it, plus sections based on that section (via extends=).

There are also a number of predefined variables: ${configname} and ${runnumber} with the obvious meanings; ${network} is the name of the network that is simulated; ${processid} and $ {datetime} expand to the OS process id of the simulation and the time it was started; and there are some more: ${runid}, ${iterationvars} and ${repetition}.

${runid} holds the Run ID. When a simulation is run, it gets assigned a Run ID, which uniquely identifies that instance of running the simulation: if you run the same thing again, it will get a different Run ID. Run ID is a concatenation of several variables like ${configname}, ${runnumber}, ${datetime} and ${processid}. This yields an identifier that is unique "enough" for all practical purposes, yet it is meaningful for humans. The Run ID is recorded into result files written during the simulation, and can be used to match vectors and scalars written by the same simulation run.

In cases when not all combinations of the iteration variables make sense or need to be simulated, it is possible to specify an additional constraint expression. This expression is interpreted as a conditional (an "if" statement) within the innermost loop, and it must evaluate to "true" for the variable combination to generate a run. The expression should be given with the constraint= configuration key. An example:

```
*.numNodes = ${n=10..100 step 10}
**.numNeighbors = ${m=2..10 step 2}
constraint = $m <= sqrt($n)
```

The expression syntax supports most C language operators (including boolean, conditional and binary shift operations) and most <math.h> functions; data types are boolean, double and string. The expression must evaluate to a boolean.

NOTE: It is not supported to refer to other iteration variables in the definition of an iteration variable (i.e. you cannot write things like ${j=$i..10}), although it might get implemented in future OMNeT++ releases.

3.3Repeating runs with different seeds

It is directly supported to perform several runs with the same parameters but different random number seeds. There are two configuration keys related to this: repeat= and seed-set=. The first one simple specifies how many times a run needs to be repeated. For example,

```
repeat = 10
```

causes every combination of iteration variables to be repeated 10 times, and the ${repetition} predefined variable holds the loop counter. Indeed, repeat=10 is equivalent of adding ${repetition=0..9} to the ini file. The ${repetition} loop always becomes the innermost loop.

The seed-set= configuration key affects seed selection. Every simulation uses one or more random number generators (as configured by the num-rngs= key), for which the simulation kernel can automatically generate seeds. The first simulation run may use one set of seeds (seed set 0), the second run may use a second set (seed set 1), and so on. Each set contains as many seeds as there are RNGs configured. All automatic seeds generate random number sequences that are far apart in the RNG's cycle, so they will never overlap during simulations.

> **NOTE:** Mersenne Twister, the default RNG of OMNeT++ has a cycle length of 2^{19937}, which is more than enough for any conceivable purpose.

The seed-set= key tells the simulation kernel which seed set to use. It can be set to a concrete number (such as seed-set=0), but it usually does not make sense as it would cause every simulation to run with exactly the same seeds. It is more practical to set it to either ${runnumber} or to ${repetition}. The default setting is ${runnumber}:

```
seed-set = ${runnumber}   # this is the default
```

This makes every simulation run to execute with a unique seed set. The second option is:

```
seed-set = ${repetition}
```

where all $repetition=0 runs will use the same seeds (seed set 0), all $repetition=1 runs use another seed set, $repetition=2 a third seed set, etc.

To perform runs with manually selected seed sets, you can just define an iteration for the seed-set= key:

```
seed-set = ${5,6,8..11}
```

In this case, the repeat= key should be left out, as seed-set= already defines an iteration and there's no need for an extra loop.

It is of course also possible to manually specify individual seeds for simulations. This is rarely necessary, but we can use it here to demonstrate another feature, parallel iterators:

```
repeat = 4
seed-1-mt = ${53542, 45732, 47853, 33434 ! repetition}
seed-2-mt = ${75335, 35463, 24674, 56673 ! repetition}
seed-3-mt = ${34542, 67563, 96433, 23567 ! repetition}
```

The meaning of the above is this: in the first repetition, the first column of seeds is chosen, for the second repetition, the second column, etc. The "!" syntax chooses the *kth* value from the iteration, where *k* is the position (iteration count) of the iteration variable after the "!". Thus, the above example is equivalent to the following:

```
# no repeat= !
seed-1-mt = ${seed1 = 53542, 45732, 47853, 33434}
seed-2-mt = ${         75335, 35463, 24674, 56673 ! seed1}
seed-3-mt = ${         34542, 67563, 96433, 23567 ! seed1}
```

That is, the iterators of seed-2-mt and seed-3-mt are advanced in lockstep with the seed1 iteration.

4 Scenarios and Result Analysis

4.1Output vectors and scalars

In OMNeT++ 3.x, the default result file names were “omnetpp.vec” and “omnetpp.sca”. This is not very convenient for batch execution, where an output vector file created in one run would be overwritten in the next run. Thus, we have changed the default file names to make them differ for every run. The new defaults are:

```
output-vector-file = "${configname}-${runnumber}.vec"
output-scalar-file = "${configname}-${runnumber}.sca"
```

This generates file names like "PureAloha-0.vec", "PureAloha-1.vec", and so on.

Also, in OMNeT++ 3.x output scalar files were always appended to by the simulation program, rather than being overwritten. This behavior was changed in 4.0 to make it consistent with vector files, that is, output scalar files are also overwritten by the simulator, and not appended to.

NOTE: The old behavior can be turned back on by setting output-scalar-file-append=true.

The way of configuring output vectors has also changed. In OMNeT++ 3.x, the keys for enabling-disabling a vector and specifying recording interval were <module-and-vectorname-pattern>.enabled, and <module-and-vectorname-pattern>.interval. The "enabled" and "interval" words changed to "enable-recording" and "recording-interval".

NOTE: The reason for this change is that per-object configuration keys are now required to have a hyphen in their names, to make it possible to tell them apart from module parameter assignments. This allows the simulator to catch mistyped config keys in ini files.

The syntax for specifying the recording interval has also been extended (in a backward compatible way) to accept multiple intervals, separated by comma. An example:

```
**.fifo[2].queueLengthVector.enable-recording = false
**.fifo[*].queueLengthVector.recording-interval = ..100, 300..500, 900..
```

Although it has nothing to do with our main topic (ini files), this is a good place to mention that the format of result files have been extended to include meta info such as the run ID, network name, all configuration settings, etc. These data make the files more self-documenting, which can be valuable during the result analysis phase, and increase reproduciblity of the results. Another change is that vector data are now recorded into the file clustered by the output vectors, which (combined with index files) allows much more efficient processing.

4.2Saving parameters as scalars

When you are running several simulations with different parameter settings, you'll usually want to refer to selected input parameters in the result analysis as well -- for example when drawing a throughput (or response time) versus load (or network background traffic) plot. Average throughput or response time numbers are saved into the output scalar files, and it is useful for the input parameters to get saved into the same file as well.

For convenience, OMNeT++ automatically saves the iteration variables into the output scalar file, so they can be referred to during result analysis. Module parameters can also be saved, but this has to be requested by the user, by configuring save-as-scalar=true for the parameters in question. The configuration key is a pattern that identifies the parameter, plus ".save-as-scalar". An example:

```
**.host[*].networkLoad.save-as-scalar = true
```

This looks simple enough. However, there are three pitfalls: non-numeric parameters, too many matching parameters, and random-valued volatile parameters.

First, the scalar file only holds numeric results, so non-numeric parameters cannot be recorded -- that will result in a runtime error.

Second, if wildcards in the pattern match too many parameters, that might unnecessarily increase the size of the scalar file. For example, if the host[] module vector size is 1000 in the example below, then the same value (3) will be saved 1000 times into the scalar file, once for each host.

```
**.host[*].startTime = 3
**.host[*].startTime.save-as-scalar = true  # saves "3" once for each host
```

Third, recording a random-valued volatile parameter will just save a random number from that distribution. This is rarely what you need, and the simulation kernel will also issue a warning if this happens.

```
**.interarrivalTime = exponential(1)
**.interarrivalTime.save-as-scalar = true  # saves random values!
```

These pitfalls are not rare in practice, so it is usually more convenient to rely on the iteration variables in the result analysis. That is, one can rewrite the above example as

```
**.interarrivalTime = exponential( ${mean=1} )
```

and refer to the $mean iteration variable instead of the interarrivalTime module parameter(s) during result analysis. save-as-scalar=true is not needed because iteration variables are automatically saved into the result files.

4.3Experiment-Measurement-Replication

We have introduced three concepts that are useful for organizing simulation results generated by batch executions or several batches of executions.

During a simulation study, a person prepares several *experiments*. The purpose of an experiment is to find out the answer to questions like *"how does the number of nodes affect response times in the network?"* For an experiment, several *measurements* are performed on the simulation model, and each measurement runs the simulation model with a different parameter settings. To eliminate the bias introduced by the particular random number stream used for the simulation, several *replications* of every measurement are run with different random number seeds, and the results are averaged.

OMNeT++ result analysis tools can take advantage of experiment/measurement/replication labels recorded into result files, and organize simulation runs and recorded output scalars and vectors accordingly on the user interface.

These labels can be explicitly specified in the ini file using the experiment=, measurement= and replication= config keys. If they are missing, the default is the following:

```
experiment = "${configname}"
measurement = "${iterationvars}"
replication = "#${repetition}, seed-set=<seedset>"
```

That is, the default experiment label is the configuration name; the measurement label is concatenated from the iteration variables; and the replication label contains the repeat loop variable and for the seed-set. Thus, for our first example the experiment-measurement-replication tree would look like this:

```
"PureAloha" -- experiment
    $N=1,$mean=0.2 -- measurement
        #0, seed-set=0 -- replication
        #1, seed-set=1
        #2, seed-set=2
        #3, seed-set=3
        #4, seed-set=4
    $N=1,$mean=0.4
        #0, seed-set=5
        #1, seed-set=6
        ...
        #4, seed-set=9
    $N=1,$mean=0.6
        #0, seed-set=10
        #1, seed-set=11
        ...
        #4, seed-set=14
    $N=2,$mean=0.2
        ...
    $N=2,$mean=0.4
        ...
    ...
```

The experiment-measurement-replication labels should be enough to reproduce the same simulation results, given of course that the ini files and the model (NED files and C++ code) haven't changed.

Every instance of running the simulation gets a unique run ID. We can illustrate this by listing the corresponding run IDs under each repetition in the tree. For example:

```
"PureAloha"
    $N=1,$mean=0.2
        #0, seed-set=0
            PureAloha-0-20070704-11:38:21-3241
            PureAloha-0-20070704-11:53:47-3884
            PureAloha-0-20070704-16:50:44-4612
        #1, seed-set=1
            PureAloha-1-20070704-16:50:55-4613
        #2, seed-set=2
            PureAloha-2-20070704-11:55:23-3892
            PureAloha-2-20070704-16:51:17-4615
        ...
```

The tree shows that ("PureAloha", "$N=1,$mean=0.2", "#0, seed-set=0") was run three times. The results produced by these three executions should be identical, unless, for example, some parameter was modified in the ini file, or a bug got fixed in the C++ code.

We believe that the default way of generating experiment-measurement-replication labels will be useful and sufficient in the majority of the simulation studies. However, you can customize it if needed. For example, here is a way to join two scenarios into one experiment:

```
[Config PureAloha_Part1]
experiment = "PureAloha"
...
[Config PureAloha_Part2]
experiment = "PureAloha"
...
```

Measurement and replication labels can be customized in a similar way, making use of named iteration variables, ${repetition}, ${runnumber} and other predefined variables. One possible benefit is to customize the generated measurement and replication labels. For example:

```
[Config PureAloha_Part1]
measurement = "${N} hosts, exponential(${mean}) packet generation interval"
```

One should be careful with the above technique though, because if some iteration variables are left out of the measurement labels, runs with all values of those variables will be grouped together to the same replications.

5 How to Migrate Ini Files

As mentioned earlier, there is no need to manually convert ini files: the IDE's ini file editor can automatically do that for you, and alternatively there is a command-line Perl script for the same purpose. This section just summarizes the conversion process for your convenience.

1. Prefix all keys in the [Cmdenv] section with "cmdenv-", and all keys in the [Tkenv] section with "tkenv-". For example, status-frequency=... becomes cmdenv-status-frequency=...
2. Move the contents of all non-Run sections into the [General] section. If there is no [General] yet, create one. The affected sections include [Cmdenv], [Tkenv], [Parameters], [OutVectors] and [Partitioning], but not [Run 1], [Run 2], etc. If there are more than one [General] sections, merge them into one.
3. Rename the [Run 1], [Run 2], etc. sections to named Config sections. If you cannot think of good names, just write [Config One], [Config Two] etc, and you can fix them up later.
4. Boolean config entries used to accept yes/no, on/off and 0/1 as values. This is no longer supported, please change any such value to true/false.
5. The following per-object config keys have been renamed, so you need to change them:

<pattern>.use-default=	☐	<pattern>.apply-default=
<pattern>.ev-output=	☐	<pattern>.cmdenv-ev-output=
<pattern>.enabled=	☐	<pattern>.enable-recording=
<pattern>.interval=	☐	<pattern>.recording-interval=
<pattern>.akaroa=	☐	<pattern>.with-akaroa=

6. The ini file is now working. However, you will probably be able to significantly reduce its length by removing entries that set module parameters to their NED default values, and replacing them with **.apply-default = true instead.

Be careful when converting INET Framework ini files: the PingApp module has parameters named "enabled" and "interval", and they are easy to confuse with output vector settings. The new rule which requires a hyphen to be present in per-object config keys will prevent such ambiguities in the future, because NED parameter names cannot contain hyphens.

NOTE: As for Step 6: Both INET Framework and Mobility Framework ini files typically contain large chunks of lines that set all sorts of module parameters to some default values. These blocks look so much alike in all ini files so that we can safely call them "boilerplate". These blocks can be all thrown out, once proper default values are entered into the framework NED files. (The 3.x NED syntax did not allow such defaults, but the redesigned 4.0 NED syntax supports it very well.)

An example:

Old

```
[General]
preload-ned-files = *.ned

[Cmdenv]
express-mode = yes
performance-display = on

[Run 1]
description="cqnB (boxed tandem Q's)"
network = cqnB
*.numTandems = 3
```

New

```
[General]
preload-ned-files = *.ned
cmdenv-express-mode = true
cmdenv-performance-display = true

[Config CQN-B]
description = "boxed tandem queues"
network = ClosedQueueingNetB
*.numTandems = 3
*.tandemQueue[*].numQueues = 5
…
```

```
*.tandemQueue[*].numQueues = 5
…
```

6
A
# ppendix: Predefined Variables

Predefined variables that can be used in configuration values:

${configname}

${runnumber}

${network}

${processid}

${datetime}

${runid}

${repetition}

${iterationvars}

${iterationvars2}

7
A
# ppendix: Configuration Keys

The following list contains all currently supported configuration keys, in alphabetical order. The list was created by passing the "–q configdetails" command-line option to a simulation executable.

> **NOTE**: This list is **not final,** and elements may appear, disappear or change until the OMNeT++ 4.0 final release.

The configuration keys:

```
<object-full-path>.cmdenv-ev-output=<bool>, default:true; per-object setting
   When cmdenv-express-mode=false: whether Cmdenv should print debug messages
   (ev<<) from the selected modules.

cmdenv-event-banner-details=<bool>, default:false; per-run setting
   When cmdenv-express-mode=false: print extra information after event
   banners.

cmdenv-event-banners=<bool>, default:true; per-run setting
   When cmdenv-express-mode=false: turns printing event banners on/off.

cmdenv-express-mode=<bool>, default:true; per-run setting
   Selects ``normal'' (debug/trace) or ``express'' mode.

cmdenv-extra-stack-kb=<int>, default:8; global setting
   Specifies the extra amount of stack (in kilobytes) that is reserved for
   each activity() simple module when the simulation is run under Cmdenv.

cmdenv-message-trace=<bool>, default:false; per-run setting
   When cmdenv-express-mode=false: print a line per message sending (by
```

send(),scheduleAt(), etc) and delivery on the standard output.

cmdenv-module-messages=<bool>, default:true; per-run setting
When cmdenv-express-mode=false: turns printing module ev<< output on/off.

cmdenv-output-file=<filename>; global setting
When a filename is specified, Cmdenv redirects standard output into the given file. This is especially useful with parallel simulation. See the `fname-append-host' option as well.

cmdenv-performance-display=<bool>, default:true; per-run setting
When cmdenv-express-mode=true: print detailed performance information. Turning it on results in a 3-line entry printed on each update, containing ev/sec, simsec/sec, ev/simsec, number of messages created/still present/currently scheduled in FES.

cmdenv-runs-to-execute=<string>; global setting
Specifies which simulation runs should be executed. It accepts a comma-separated list of run numbers or run number ranges, e.g. 1,3-4,7-9. If the value is missing, Cmdenv executes all runs that have ini file sections; if no runs are specified in the ini file, Cmdenv does one run. The -r command line option overrides this setting.

cmdenv-status-frequency=<int>, default:100000; per-run setting
When cmdenv-express-mode=true: print status update every n events. Typical values are 100,000...1,000,000.

configuration-class=<string>; global setting
Part of the Envir plugin mechanism: selects the class from which all configuration will be obtained. This option lets you replace omnetpp.ini with some other implementation, e.g. database input. The simulation program still has to bootstrap from an omnetpp.ini (which contains the configuration-class setting). The class has to implement the cConfiguration interface.

constraint=<string>; per-run setting
For scenarios. Contains an expression that iteration variables (${} syntax) must satisfy for that simulation to run. Example: $i < $j+1.

cpu-time-limit=<double>, unit="s"; per-run setting
Stops the simulation when CPU usage has reached the given limit. The default is no limit.

debug-on-errors=<bool>, default:false; global setting
When set to true, runtime errors will cause the simulation program to break into the C++ debugger (if the simulation is running under one, or just-in-time debugging is activated). Once in the debugger, you can view the stack trace or examine variables.

description=<string>; per-run setting
Descriptive name for the given simulation configuration. Descriptions get displayed in the run selection dialog.

<object-full-path>.enable-recording=<bool>, default:true; per-object setting
Whether data written into an output vector should be recorded.

eventlog-file=<filename>; per-run setting

Name of the event log file to generate. If empty, no file is generated.

eventlog-message-detail-pattern=<custom>; global setting
A list of patterns separated by '|' character which will be used to write message detail information into the event log for each message sent during the simulation. The message detail will be presented in the sequence chart tool. Each pattern starts with an object pattern optionally followed by ':' character and a comma separated list of field name patterns. In the object pattern and/or/not/* and various field matcher expressions can be used. The field pattern contains a wildcard expressions matched against field names.

experiment=<custom>, default:${configname}; per-run setting
Experiment label. This string gets recorded into result files, and may be referred to during result analysis.

extends=<string>; per-run setting
Name of the configuration this section is based on. Entries from that section will be inherited and can be overridden. In other words, configuration lookups will fall back to the base section.

fingerprint=<string>; per-run setting
The expected fingerprint, suitable for crude regression tests. If present, the actual fingerprint is calculated during simulation, and compared against the expected one.

fname-append-host=<bool>, default:false; global setting
Turning it on will cause the host name and process Id to be appended to the names of output files (e.g. omnetpp.vec, omnetpp.sca). This is especially useful with distributed simulation.

ini-warnings=<bool>, default:false; global setting
Currently ignored. Accepted for backward compatibility.

load-libs=<filenames>; global setting
Specifies dyamic libraries to be loaded on startup. The libraries should be given without the `.dll' or `.so' suffix -- that will be automatically appended.

<object-full-path>.max-buffered-samples=<int>; per-object setting
For output vectors: the maximum number of values to buffer per vector, before writing out a block into the output vector file.

measurement=<custom>, default:${iterationvars}; per-run setting
Measurement label. This string gets recorded into result files, and may be referred to during result analysis.

network=<string>; per-run setting
The name of the network to be simulated.

num-rngs=<int>, default:1; per-run setting
Number of the random number generators.

output-scalar-file=<filename>, default:${configname}-${runnumber}.sca; per-run setting
Name for the output scalar file.

output-scalar-file-append=<bool>, default:false; per-run setting
What to do when the output scalar file already exists: append to it

(OMNeT++ 3.x behavior), or delete it and begin a new file (default).

output-scalar-precision=<int>, default:14; per-run setting
Adjusts the number of significant digits for recording numbers into the
output scalar file.

output-vector-file=<filename>, default:${configname}-${runnumber}.vec; per-run setting
Name for the output vector file.

output-vector-precision=<int>, default:14; per-run setting
Adjusts the number of significant digits for recording numbers into the
output vector file.

output-vectors-memory-limit=<double>, unit="B", default:16MB; per-run setting
Total memory that can be used for buffering output vectors. Larger values
produce less fragmented vector files (i.e. cause vector data to be grouped
into larger chunks), and therefore allow more efficient processing later.

outputscalarmanager-class=<string>, default:cFileOutputScalarManager; global setting
Part of the Envir plugin mechanism: selects the output scalar manager class
to be used to record data passed to recordScalar(). The class has to
implement the cOutputScalarManager interface.

outputvectormanager-class=<string>, default:cIndexedFileOutputVectorManager; global setting
Part of the Envir plugin mechanism: selects the output vector manager class
to be used to record data from output vectors. The class has to implement
the cOutputVectorManager interface.

parallel-simulation=<bool>, default:false; global setting
Enables parallel distributed simulation.

parsim-communications-class=<string>, default:cFileCommunications; global setting
If parallel-simulation=true, it selects the class that implements
communication between partitions. The class must implement the
cParsimCommunications interface.

parsim-synchronization-class=<string>, default:cNullMessageProtocol; global setting
If parallel-simulation=true, it selects the parallel simulation algorithm.
The class must implement the cParsimSynchronizer interface.

<object-full-path>.partition-id=<int>; per-object setting
With parallel simulation: in which partition the module should be
instantiated.

perform-gc=<bool>, default:false; global setting
Whether the simulation kernel should delete on network cleanup the
simulation objects not deleted by simple module destructors. Not
recommended.

preload-ned-files=<filenames>; global setting
NED files to be loaded dynamically. Wildcards, @ and @@ listfiles accepted.

print-undisposed=<bool>, default:true; global setting
Whether to report objects left (that is, not deallocated by simple module
destructors) after network cleanup.

realtimescheduler-scaling=<double>; global setting
When cRealTimeScheduler is selected as scheduler class: ratio of simulation time to real time. For example, scaling=2 will cause simulation time to progress twice as fast as runtime.

<object-full-path>.record-event-numbers=<bool>, default:true; per-object setting
Whether to record event numbers for an output vector. Simulation time and value are always recorded. Event numbers are needed by the Sequence Chart Tool, for example.

<object-full-path>.recording-interval=<custom>; per-object setting
Recording interval(s) for an output vector. Syntax: [<from>]..[<to>], …
Examples: 100..200, 100.., ..200

repeat=<int>, default:1; per-run setting
For scenarios. Specifies how many replications should be done with the same parameters (iteration variables). This is typically used to perform multiple runs with different random number seeds. The loop variable is available as ${repetition}. See also: seed-set= key.

replication=<custom>, default:#${repetition}, seedset=@; per-run setting
Replication label. This string gets recorded into result files, and may be referred to during result analysis.

rng-class=<string>, default:cMersenneTwister; per-run setting
The random number generator class to be used. It can be `cMersenneTwister', `cLCG32', `cAkaroaRNG', or you can use your own RNG class (it must be subclassed from cRNG).

<object-full-path>.save-as-scalar=<bool>, default:false; per-object setting
Applicable to module parameters: specifies whether the module parameter should be recorded into the output scalar file. Set it for parameters whose value you'll need for result analysis.

scheduler-class=<string>, default:cSequentialScheduler; global setting
Part of the Envir plugin mechanism: selects the scheduler class. This plugin interface allows for implementing real-time, hardware-in-the-loop, distributed and distributed parallel simulation. The class has to implement the cScheduler interface.

seed-%-lcg32=<int>; per-run setting
When cLCG32 is selected as random number generator: seed for the kth RNG. (Substitute k for '%' in the key.)

seed-%-mt=<int>; per-run setting
When Mersenne Twister is selected as random number generator (default): seed for RNG number k. (Substitute k for '%' in the key.)

seed-%-mt-p%=<int>; per-run setting
With parallel simulation: When Mersenne Twister is selected as random number generator (default): seed for RNG number k in partition number p. (Substitute k for the first '%' in the key, and p for the second.)

seed-set=<int>, default:${runnumber}; per-run setting
Selects the kth set of automatic random number seeds for the simulation. Two meaningful values are ${repetition} which is the repeat loop counter (see repeat= key), and ${runnumber}.

sim-time-limit=<double>, unit="s"; per-run setting
Stops the simulation when simulation time reaches the given limit. The default is no limit.

simtime-scale=<int>, default:-12; global setting
Sets the scale exponent, and thus the resolution of time for the 64-bit fixed-point simulation time representation. Accepted values are -18..0; for example, -6 selects microsecond resolution. -12 means picosecond resolution, with a maximum simtime of ~110 days.

snapshot-file=<filename>, default:${configname}-${runnumber}.sna; per-run setting
Name of the snapshot file.

snapshotmanager-class=<string>, default:cFileSnapshotManager; global setting
Part of the Envir plugin mechanism: selects the class to handle streams to which snapshot() writes its output. The class has to implement the cSnapshotManager interface.

tkenv-anim-methodcalls=<bool>, default:true; global setting
Whether to animate method calls across modules. Only calls to methods which contain Enter_Method() can be animated.

tkenv-animation-enabled=<bool>, default:true; global setting
Enables/disables animation.

tkenv-animation-msgclassnames=<bool>, default:true; global setting
Enables/disables displaying the C++ class name of messages during animation.

tkenv-animation-msgcolors=<bool>, default:true; global setting
Enables/disables using different colors for each message kind during message flow animation.

tkenv-animation-msgnames=<bool>, default:true; global setting
Enables/disables displaying message names during message flow animation.

tkenv-animation-speed=<double>, default:1.5; global setting
Controls the speed of the animation; values in the range 0..3 are accepted.

tkenv-default-config=<string>; global setting
Specifies which config Tkenv should set up automatically on startup. The default is to ask the user.

tkenv-default-run=<int>, default:0; global setting
Specifies which run (of the default config, see tkenv-default-config) Tkenv should set up automatically on startup. The default is to ask the user.

tkenv-expressmode-autoupdate=<bool>, default:true; global setting
Enables/disables updating the inspectors during Express mode.

tkenv-extra-stack-kb=<int>, default:48; global setting
Specifies the extra amount of stack (in kilobytes) that is reserved for each activity() simple module when the simulation is run under Tkenv.

tkenv-image-path=<filename>; global setting
Specifies the path for loading module icons.

tkenv-methodcalls-delay=<double>, unit="s", default:0.2; global setting
Sets delay after method call animation.

tkenv-next-event-markers=<bool>, default:true; global setting
Whether to display the next event marker (red rectange).

tkenv-penguin-mode=<bool>, default:false; global setting
Surprise surprise.

tkenv-plugin-path=<filename>; global setting
Specifies the search path for Tkenv plugins. Tkenv plugins are .tcl files that get evaluated on startup.

tkenv-print-banners=<bool>, default:true; global setting
Enables/disables printing banners for each event.

tkenv-senddirect-arrows=<bool>, default:true; global setting
Whether to display arrows during animation of sendDirect() calls.

tkenv-show-bubbles=<bool>, default:true; global setting
Whether to honor the bubble() calls during animation.

tkenv-show-layouting=<bool>, default:false; global setting
Show layouting process of network graphics.

tkenv-slowexec-delay=<double>, unit="s", default:0.3; global setting
Delay between steps when you slow-execute the simulation.

tkenv-update-freq-express=<int>, default:10000; global setting
Number of events executed between two display updates when in Express execution mode.

tkenv-update-freq-fast=<int>, default:50; global setting
Number of events executed between two display updates when in Fast execution mode.

tkenv-use-mainwindow=<bool>, default:true; global setting
Enables/disables writing ev output to the Tkenv main window.

tkenv-use-new-layouter=<bool>, default:false; global setting
Selects between the new and the old (3.x) network layouting algorithms.

total-stack-kb=<int>, default:0; global setting
Specifies the maximum memory for activity() simple module stacks in kilobytes. You need to increase this value if you get a ``Cannot allocate coroutine stack'' error.

user-interface=<string>; global setting
Selects the user interface to be started. Possible values are Cmdenv and Tkenv, provided the simulation executable contains the respective libraries or loads them dynamically.

warnings=<bool>, default:true; per-run setting
Enables warnings.