QCMSim Quick Reference Guide

Last update: 01.15.2014

Contents

[Simulator Environment 2](#_Toc377545203)

[Introduction 2](#_Toc377545204)

[Top Layer: QCN, Seismic Models, and QCN Explorer 2](#_Toc377545205)

[Middle Layer: Computer Network Components 3](#_Toc377545206)

[Bottom Layer: Discrete Event-based Queuing System Simulator 3](#_Toc377545207)

[Simulator Implementation 4](#_Toc377545208)

[Terms in QCNSim 5](#_Toc377545209)

[Token 5](#_Toc377545210)

[Protocol Data Unit (PDU) 5](#_Toc377545211)

[Facility 5](#_Toc377545212)

[Traffic Generator 5](#_Toc377545213)

[Sensor 5](#_Toc377545214)

[Node 5](#_Toc377545215)

[Route 6](#_Toc377545216)

[Link 6](#_Toc377545217)

[Terms in QCN Explorer 6](#_Toc377545218)

[Seismic Event Generator 6](#_Toc377545219)

[Client Node 6](#_Toc377545220)

[Intermediate Node 6](#_Toc377545221)

[Destination node 7](#_Toc377545222)

[Constructing a Simulation 7](#_Toc377545223)

[Event Handling 7](#_Toc377545224)

[Documentation 9](#_Toc377545225)

[Changelog 9](#_Toc377545226)

[References 9](#_Toc377545227)

# Simulator Environment

## Introduction

Our simulation environment allows us to study the QCN cyber-infrastructure including an advanced network topology and multiple QCN servers that support each other in case of network failures or overload. In particular, our environment is capable of constructing topologies in which sensors are connected to multiple QCN servers via any number of links and intermediate network nodes. Modeling the links includes infinite or limited-size transmission queues (buffers) and propagation delay for diverse media (e.g., copper wire, fiber optics, and air). Additionally, the environment supports dynamic behavior for the topology, in which sensors can reroute their traffic and links can be shut down during a simulation.

The simulation environment is composed of three layers and is based in part on EmBOINC (Estrada et al. 2009) for QCN and the TARVOS Computer Networks Simulator (Portnoi and Martins 2007). The three layers are represented in Figure 1; each implements different levels of abstraction for the QCN simulation and its components. We describe the layers in further detail in the next sections.

## Top Layer: QCN, Seismic Models, and QCN Explorer

The top layer integrates the scientific models of earthquake events and sensors. A generic seismology model includes the two seismic waves called P- and S-waves that propagate outward from the hypocenter (the location that an earthquake starts beneath the surface of the Earth) in a spherical pattern. The speed of the S-wave is generally around 3 km/s, but can vary in different regions as seismic waves travel slower through looser substrates and faster through denser ones. The speed of the P-wave, or primary wave, is modeled as , where is the speed of the S-wave. The amplitude of the seismic waves, measured in m/s2, is a measurement obtained by the QCN accelerometers and is used to quantify the shaking effects at the sensor location. The amplitude of the seismic waves is calculated from the attenuation relationship described in (Chung et al. 2011). The amplitude of the S-wave is based on the sensor’s hypocentral distance (i.e., the straight-line distance within the Earth between the earthquake’s hypocenter and the sensor’s location on the surface). The amplitude of the P-wave is assumed to be one fifth that of the S-wave. Because the goal of this layer is to offer high-level abstractions to augment functionalities of the underlying layers and to assist the user in constructing seismic simulations, this layer also comprises configuration file parsers and the QCN Explorer (Schlachter, Benson, and Estrada 2013) user interface.

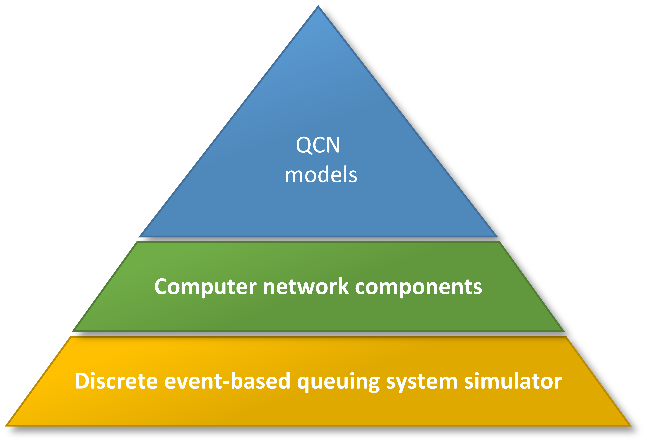


Figure : The simulation environment’s three-layer construction structure.

QCN Explorer has a web-based interface and provides graphical tools to the user to prepare earthquake simulation scenarios by controlling earthquake parameters and placement of QCN sensors as well as drawing the network topology on a real-life geographic map. Through its earthquake simulation models, QCN Explorer generates seismic events as input to the top layer file parsers.

## Middle Layer: Computer Network Components

The middle layer provides the QCN cyber-infrastructure with the high-level abstractions for computer network components, such as network links, Protocol Data Units (PDUs), nodes, routing, link failures, and traffic generators. The layer also includes components to measure performance, calculate statistics, and generate traces. Classes in this layer utilize kernel classes from the lower layer to perform their duties. The main classes are Node, Link, ProtocolDataUnit, and Route.

In the QCN cyber-infrastructure, nodes implemented in the Node class are either BOINC clients connected to low cost sensors, or intermediary entities (such as intermediate network nodes and links). Nodes receive trickle messages from sensors or links, forward them to other nodes through links, take decisions on paths and routes, and discard trickle messages. Nodes also collect statistics such as delay, jitter, and the number of received, forwarded, discarded trickle message and bytes, behaving much like network routers. The Link class implements connections between BOINC clients and intra-topology nodes, intra-topology nodes and a QCN server, or a BOINC client and a QCN server. More specifically, links connects two QCN nodes in one way (i.e., simplex links). In more complex frameworks than QCN, this class also supports two-way connections (i.e., duplex links that are de facto two simplex links). Links provide service by transporting trickle messages from one end of the link to the other end (e.g., from a BOINC client to a QCN intra-topology node or between two intra-topology node). This is done in two steps. First, upon being requested, the link first transmits the trickle messages at full bandwidth speed; the time to transmit a message is a relationship between message length and bandwidth speed. If transmission is successful, the message is considered within the link’s medium (e.g., the wire or airborne for a wireless link). Second, the message propagates through the medium until reaching the QCN server. This propagation is simulated as a simple fixed delay defined by the user based on QCN traces. Consequently, a network link is modeled as a resource with one QCN server and a priority queue that is chained to the infinite-capacity QCN server. The ProtocolDataUnit class models a network PDU representing a QCN trickle message. In particular, this class associates Time-To-Live (TTL) and length members to PDUs. The Route class models routes within a QCN network of intra-nodes, to be followed by PDUs.

## Bottom Layer: Discrete Event-based Queuing System Simulator

The bottom layer is a general-purpose, discrete event-driven queuing system (Jain 1991) simulator. This layer provides the classes used to implement the higher layer components that allow the construction of the advanced network topologies proposed for QCN in this paper. The layer is composed by a kernel including elements, such as resources that provide service, priority and FIFO queues for the resources, functions for event manipulation, and statistics collection and traffic generators based on probability distributions.

The kernel’s view is a queuing system with three main components: resources, tokens, and events. Resources and tokens are part of a parent class in the simulator called Entity. These resources model real-life entities, such as routers, links, web servers, and processors that basically deliver some sort of service. Resources have a series of related functions or methods (e.g., definition, reservation, release, preemption, and fail). The simulator defines a resource object by a reference and the number of processing units within that resource. Processing units work independently from the other units within the same resource, and they share a single priority queue. The number of processing units may model, for example, cores within a processor, allowing for modeling threads or parallel processing of clients or web requests. When service is requested, a reservation is submitted for a resource to perform the service. If the resource has at least one free processing unit, then service is granted and that processing unit becomes busy. When the service is finished, that unit is released and becomes free. If a resource has no free processing units, then the resource is busy; otherwise, the resource is free if at least one of its processing units is free. In addition, the kernel offers capabilities for making a resource temporarily non-operational. This ability can be used to model failures, such as a failed link in the QCN network or an unavailable QCN server.

The tokens represent entities in the system (e.g., customers, tasks, Protocol Data Units or PDUs, and messages). The flow of tokens through the collection of resources models the dynamic behavior of a system; in particular, this flow models the QCN cyber-infrastructure. A token class depicts the several data structures and functions associated with tokens, including an associated object that can be used to carry trickle messages across resources. Resources, therefore, perform service on the tokens. In a queuing system simulation context, a service is merely a delay; real processing of the token or associated data is not generally done. However, upper simulator layers, such as the Top Layer described earlier, do implement classes to perform processing in the associated token data (e.g., to obtain seismic analysis).

Finally, an event defines any change of state in the system. Changes of state include, for instance, the generation of a new token, a service request for a token, a resource release, dequeuing of a token, and a subsequent request for service. Events have an occurrence time, the type of event, and an associated entity object that is used by the simulator to manipulate the event.

A class called Scheduler is the main kernel class that coordinates interaction between the simulator components and controls the simulation clock. In contrast with a real-time application or an emulator, the simulator works under simulated time, by simulating the passage of real time in a generally faster fashion. In our event-driven simulator, this is achieved by generating events that occur in specific times in the future and putting these events into the Event Chain class. Event Chain is a time-ordered list of events. The Scheduler class is the entity responsible for putting events into the list, ordered by occurrence in time, and retrieving events from the beginning of the list. When the Scheduler retrieves an event at the head of the list, it causes the event; this activates the agents responsible for executing the actions predicted in the event. For instance, an arrival event, which can be described as a token “arriving” at a specified destination, can trigger a request-for-service event. To start properly, a simulation requires at least one initial event in the Event Chain. This initial event schedules other events generating a chain, and the simulation proceeds by adding the events in the Event Chain. If there are no more events in the Event Chain, then the simulation terminates. The simulation can also stop due to other conditions (e.g., when the simulation reaches the targeted simulation time).

## Simulator Implementation

The simulation environment is coded in C++11, the version of C++ approved by ISO on August 12, 2011. This C++ standard offers modern coding resources, namely controlled type inference; range-based loops resembling Java’s and Python’s capabilities; constructor delegation; constructor initialization lists; abstract classes; robust general-purpose smart pointers. allowing the compiler to deal with heap memory management and thus avoid dangerous memory-leak situations; and extensible random number generators, including several probability distributions, such as Poisson, exponential, and normal. By using this C++ standard, the programmer has stronger support to generate streamlined code and, in particular, can allocate more effort on the purposes of the code and less on language-specific issues like heap memory management. The coding employs the test-driven development technique. As benefits from this method, the test cases employed constitute a valuable piece of documentation, and development can progress in a solid pace as larger components of the code are assured to rely on smaller tested and proven components. We utilized solely standard C++11 libraries in the implementation of our tool, and no platform-specific customization. This way, the only basic requirement for running the tool in any computing platform is a C++11-compatible compiler.

# Terms in QCNSim

## Token

Tokens represents the active entities of the system. The dynamic behavior of the system is modeled by the movement of tokens through set of facilities. A token may represent a task in a computer system model, a packet in communication model or memory access in a memory bus subsystem model. A token may reserve (preempt) a facility or schedule activity of various durations. A token can be a single integer (customer id), an structure, (enter time, size, etc.) or another object (such as a packet).

The routing of tokens throughout the simulator is done by carefully constructing the chain of events (which event schedules with event) and by putting pertinent information within the token, i.e., the source Entity and destination Entity, or previous Entity, next Entity. Upon processing an event, a processing function can capture, for example, the appropriate facility to which request service for a token. This class, the Route class, and the Node class implement routing resources.

## Protocol Data Unit (PDU)

This class inherits from Token class and models a PDU or Protocol Data Unit. This class will extend Token and add features for computer networks, such as TTL (Time To Live) and size.

## Facility

The Facility class is used to model a resource or service center, i.e., something from which a process requests service. A facility consists of one or more servers and one priority queue, in which processes await access to the next available server. In a high-level, a facility will model a network link and possibly a BOINC server, if this BOINC server does any processing.

## Traffic Generator

The Traffic Generator class is a parent class for classes that generate traffic for the simulator. A traffic generator is typically an event generator, which generates events in time distributed according to some probability distribution. For instance, we would have exponential traffic generator, constant traffic generator, Pareto traffic generator, Gaussian traffic generator, random traffic generator.

Traffic generators are attached to Nodes.

Tokens or PDUs generated by the Traffic Generators will follow an explicit route, which will be attached to each Token/PDU.

## Sensor

The Sensor class inherits from Traffic Generator. This class, however, generates traffic based on seismic events. In particular, seismic events are primarily populated into an event list or event chain. When a seismic event reaches a certain Sensor, this sensor will generated a traffic event, essentially a Protocol Data Unit (or message) directed to a certain BOINC server.

As a Traffic Generator, sensors are attached to Nodes.

## Node

The Node class describes several statistics for counting PDUs and bytes received, for instance, and to measure jitter and delay for PDUs. Typically, a node measures statistics for network layer (network PDU, therefore). Entities can be "attached" to nodes: these entities are typically traffic generators (sources), links, and application servers.

This simulator works by abstracting a computer network into traffic generators, nodes, links, and possibly sinks and application servers. A traffic generator is attached to a node, which servers as the source of the PDU/token. The generated PDU/token will have a destination node, and this node will typically be the sink for the PDU. An application server may be connected to a node. This application server Entity may receive the contents of the PDU/token and process it, generating some result or more PDUs/tokens. A node with a sink simply means that node is the end destination of the PDU/token, and the PDU/token will be dropped upon receipt, after proper statistics updated. Links will be attached to nodes, and each link connects two nodes. There can be many links attached to a single node. The node will perform PDU/token forwarding based on some routing rule. Forwarding consists of consulting a routing table or rule upon receipt of a PDU/token, if that node is not the final destination of the PDU/token. The routing table or rule will define the next link (attached to the current node) through which the PDU/token will be forwarded. Note that this "next link" is similar to the "next hop" information on typical routing tables. "Next link," "attached application server," "traffic generators" are all Entity objects.

## Route

This class models routes within a network, to be followed by tokens or by PDUs. A route is basically a list of Entity objects a token will follow throughout the network of facilities, in a low-level point of view. In a high-level point of view, a token/PDU will follow nodes and links, each of these Entities. A typical route will indicate nodes only, however. High-level functions must abstract the nodes and populate links to complete a route.

## Link

A network link connects two nodes, and can be unidirectional (simplex) or bidirectional (duplex). A link has, as parameters, bandwidth, propagation delay, source node and destination node (in duplex links, these nodes can be considered interchangeable). A duplex link is constructed, in QCNSim, as a (forward) link object that has a pointer to another (simplex, reverse) link object within. This embedded (simplex, reverse) link object does not point back to the forward link object.

# Terms in QCN Explorer

These are potential terms to be used in QCN Explorer when it is integrated with QCNSim. There are listed here for reference.

## Seismic Event Generator

This generator will produce seismic events, which in turn will integrate the event chain within the QCNSim backend.

## Client Node

A client Node, in the Web Interface, represents a Sensor attached to a Node for the backend.

Each sensor that is placed in Sam's simulator needs to create a node to which it is linked (this node will be in the same location as the sensor). Each client node must also have a route stored that it will use to send information to a destination node.

The route must be defined by the user. Traffic (or messages) generated by Client Nodes will contain this route and will be passed to the backend.

A Client Node is connected to an Intermediate Node (i.e., the backend Node within the client node is connected to the backend Node within the intermediate node).

Regions can be designated by the user to help connect several Client Nodes to one Intermediate Node.

## Intermediate Node

An Intermediate Node, in the Web Interface, is simply a Node for the backend.

Each Intermediate Node is connected to one or more Intermediate Nodes through network links. A region can be designated, in which all Client Nodes inside that region are linked to one Intermediate Node.

The connections between Intermediate Nodes and Client nodes will produce a route, which will be attached to respective Client Nodes.

## Destination node

A Destination Node, in the Web Interface, is simply a Node for the backend. They will have BOINC servers attached.

A Destination Node is connected to one or more Intermediate Nodes, or Client Nodes, via links.

# Constructing a Simulation

A typical simulation basically consists of creating a network topology, creating traffic generators, handling events and generating traces or reports during, or after the simulation is concluded. The process is summarized below:

1. Declare/define variables and data structures to handle objects and other desired statistics.
2. Create network topology:
   1. Create the Node objects.
   2. Create the ExplicitRoute objects.
   3. Create Link objects (duplex and/or simplex links).
   4. Create traffic generators, typically QcnSensorTrafficGenerator objects.
3. Create initial events.
4. Handle events within a while/switch/case structure.
5. Generate statistics/traces.

# Event Handling

Event handling consists of, within a while loop that lasts until the simulation is over, fetching the next event from the event chain (through the Scheduler object) and handling the event according to its type. In general, handling the event will generate another event (this the chaining of events), either by directly scheduling the next event, or by calling simulator functions that schedule the next event automatically.

The general structure to follow for event handling is depicted in Algorithm 1.

Algorithm : Basic event handling structure.

|  |
| --- |
| **while** simulation is not concluded;  Fetch next event from Scheduler (event chain);  Get embedded *PDU* from event;  **switch** *EventType*;  **case** BEGIN\_SIMULATION: // Optional.  Initiate tracing utilities;  Print initial messages or report headers;  Schedule TURN\_ON\_GENERATORS;  **end case;**  **case** TURN\_ON\_GENERATORS; // Optional.  Turn on specific traffic generators;  Create instance of traffic generator event;  // Schedules event TRAFFIC\_GENERATOR\_ARRIVAL.  **end case;**  **case** TURN\_OFF\_GENERATORS: // Optional.  Turn off specific traffic generators;  **end case;**  **case** SEISMIC\_EVENT\_DETECTION:  Get seismic event data;  Create an instance of QCN traffic generator event for appropriate sensor;  // Schedules event TRAFFIC\_GENERATOR\_ARRIVAL.  **end case;**  **case** TRAFFIC\_GENERATOR\_ARRIVAL:  Schedule event PDUTOKEN\_ARRIVAL\_AT\_NODE for *PDU*;  Treat specific cases, such as traffic rerouting;  **end case;**  **case** PDUTOKEN\_ARRIVAL\_AT\_NODE:  Find *link* that connects previous node to next node within *PDU*;  **if** *link* is found **then**  End *PDU* propagation at *link*;  **else**  Do nothing; this is the source node;  **end if;**  Process and forward the *PDU;*  **if** current node is not the destination node **then**  schedule event REQUEST\_PDU\_TRANSMISSION\_AT\_LINK;  **else**  // This is the destination node.  Output tracing data to file;  Deliver *PDU* to BOINC server facility; // Optional.  **end if;**  **end case;**  **case** REQUEST\_PDU\_TRANSMISSION\_AT\_LINK:  Find *link* that connects previous node to next node within *PDU*;  Transmit *PDU* to this *link*;  // Schedules event END\_TRANSMISSION\_PROPAGATE\_PDU\_AT\_LINK.  **end case;**  **case** END\_TRANSMISSION\_PROPAGATE\_PDU\_AT\_LINK:  Find *link* that connects previous node to next node within *PDU*;  Propagate *PDU* through this *link*;  // Schedules event END\_PROPAGATION\_AT\_LINK.  **end case;**  **case** END\_PROPAGATION\_AT\_LINK:  Schedule event PDUTOKEN\_ARRIVAL\_AT\_NODE;  **end case;**  **case** END\_SIMULATION:  Signal end of simulation to while loop (make a flag **true**);  **end case;**  **case** SET\_LINK\_DOWN: // Optional.  Set specific link down;  **end case;**  **case** SET\_LINK\_DOWN: // Optional.  Set specific link down;  **end case;**  **case** SET\_LINK\_UP: // Optional.  Set specific link up;  **end case;**  **end switch;**  **end while;** |

# Documentation

The simulator code is fully documented with inline comments and Doxygen-style tag comments. To view the Doxygen-generated comments, go to ~/doc/html/index.html.

To generate or update Doxygen documentation, there is a configuration file at the root directory of the simulator’s project, at ~/Doxyfile. The configuration is essentially pointing to the proper directories in which the source files are located and some Doxygen-specific options.

# Changelog

* **20140102**: Complete unit tests and results. Suggestion: implement Topology class to accrue data structures related to network topology (vectors of nodes, vector of links, functions to find a link given two nodes, etc. See QcnSimCCGrid.cpp for examples.
* **20131121**: Tagging version utilized to generate CCGrid paper results.
* **20130828**: Ready to generate results for example scenarios. To do: Link class, seismic models. XML parameter reader. Trace/log functions.
* **20130809**: Tagging revision 337, 20130809 for demo. Most kernel functionality implemented. Simulating topology in which client traffic is rerouted to backup server mid-simulation.

# References

Chung, A. I., C. Neighbors, A. Belmonte, M. Miller, H. H. Sepulveda, C. Christensen, R. S. Jakka, E. S. Cochran, and J. F. Lawrence. 2011. "The Quake-Catcher Network Rapid Aftershock Mobilization Program Following the 2010 M 8.8 Maule, Chile Earthquake." *Seismological Research Letters* no. 82 (4):526-532.

Estrada, T., M. Taufer, K. Reed, and D.P. Anderson. 2009. EmBOINC: An emulator for performance analysis of BOINC projects. Paper read at Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, May.

Jain, Raj K. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. 1 ed: Wiley- Interscience.

Portnoi, Marcos, and Joberto Sergio Barbosa Martins. 2007. TARVOS - an Event-Based Simulator for Performance Analysis, Supporting MPLS, RSVP-TE, and Fast Recovery. Paper read at XIII Brazilian Symposium on Multimedia and the Web - Webmedia 2007, October 20-24, at Gramado, RS, Brazil.

Schlachter, Samuel, Kyle Benson, and Trilce Estrada. 2013. QCN Explorer. <http://qcnexplorer.org/>.