Einführung in OMNeT++ und OverSim

Till Knollmann

15th May 2017

Contents

1 Was ist OMNeT++ / OverSim? 3

2 Das Modell von OMNeT++ / OverSim 3
2.1 Wie simuliert OMNeT / OverSim? . . . 3
2.2 Erweiterungen für VADs . . . 4

3 Einrichtung 4
3.1 Einrichtung ohne virtuelle Maschine . . . 5
3.1.1 Herunterladen der benötigten Dateien . . . 5
3.1.2 Einrichten der OMNeT++ IDE . . . 6
3.1.3 Einbinden von OverSim . . . 6
3.1.4 Einbinden der Erweiterungen . . . 6
3.1.5 Konfigurieren des Compilers . . . 6
3.2 Einrichten als virtuelle Maschine . . . 7
3.2.1 Einrichten . . . 7
3.2.2 Erstellen eines geteilten Ordners . . . 10

4 OMNeT++ IDE 12
4.1 Übersicht . . . 12
4.2 Oberfläche während der Simulation . . . 13

5 Einführungsbeispiele 15
5.1 Hello . . . 15
5.2 Chain . . . 15
5.3 Clique . . . 16
5.4 Broadcast . . . 17

6 Beispiel: Einfache Kette 18
6.1 Anlegen des neuen Netzwerkes . . . 19
6.2 Definieren der Overlay-Logik . . . 20
6.3 Definieren von neuen Nachrichten . . . 25
6.4 Konfigurationen anlegen . . . 26
6.5 Ausführen der Simulation . . . 27

7 Hilfreiche Ressourcen 27

8 OMNeT++ Install Guide 28

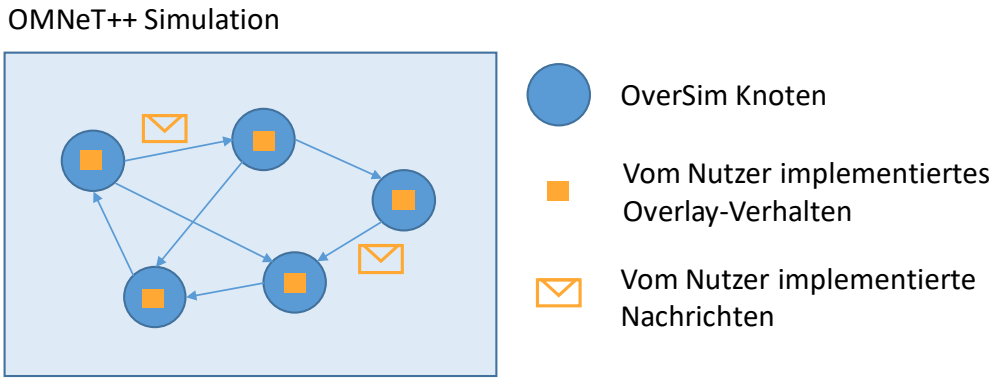
1 Was ist OMNeT++ / OverSim?

OMNeT++ ist eine ereignisbasierte Simulationsumgebung und ein Framework, um einfach und schnell Netzwerksimulationen erstellen und auswerten zu können. OMNeT++ basiert auf C++ und ist Komponentenweise aufgebaut. Es werden sowohl Linux als auch Windows und Mac OS unterstützt.
OverSim wurde am Karlsruhe Institute of Technology entwickelt und ist ein eigenes Framework speziell zum Auswerten und Simulieren von Overlay-Netzwerken. Dieses Framework baut auf OMNeT++ auf und bietet viele Funktionen um Simulationen von Overlays einfach und anschaulich durchführen zu können.

2 Das Modell von OMNeT++ / OverSim

2.1 Wie simuliert OMNeT / OverSim?

Simulationen in OMNeT++ sind ereignisbasiert. Ein Ereignis ist dabei stets eine ankommende Nachricht. OMNeT++ simuliert, indem es schrittweise eine verfügbare Nachricht auswählt und diese an ihr Ziel ausliefert. Jedes Element einer OMNeT++ Simulation ist dabei ein so genanntes Modul. Ein Modul kann aus mehreren Submodulen bestehen. (Sub-)Module sind untereinander durch Connections verbunden. Der Nutzer kann nun eigene Module und Submodule definieren und diese beliebig durch Connections verknüpfen. Die Definition von Modulen erfolgt in sogenannten .ned-Dateien. Zusätzlich wird zu einem Modul eine C++ Klasse generiert, in der die Logik des Moduls implementiert werden kann. Der Nutzer kann auch eigene Nachrichten implementieren. Die Definition dieser Nachrichten erfolgt in .msg-Dateien.



OverSim bietet bereits vorgefertigte Module und Mechanismen, um Overlay-Netzwerke einfach simulieren zu können. Als Nutzer können wir unser eigenes Overlay-Netzwerk und die darin verankerte Logik implementieren, indem wir von den vorgefertigten Modulen erben lassen. OverSim bettet beim Simulieren dann die vom Nutzer erstellten Module und ihre Logik so ein, dass sie wie ein typisches Overlay-Netzwerk simuliert

werden können. Das Simulieren der Verknüpfungen zwischen den Modulen und der Nachrichtenübertragung wird komplett von OverSim übernommen.

2.2 Erweiterungen für VADs

Zum leichteren Einstieg wurden einige Beispiele auf Basis von Erweiterungsklassen und -modulen implementiert. Durch diese Erweiterungen wird das Implementieren einer eigenen Overlay-Struktur weiter vereinfacht, indem jede Logik in die folgenden automatisch aufgerufenen Methoden implementiert werden kann:

- *onInitialize()*: Wird aufgerufen, wenn ein Overlay-Knoten initialisiert wird.
- *onTimeout()*: Wird regelmäßig an einem Overlay-Knoten aufgerufen.
- *onMessageReceived()*: Wird aufgerufen, wenn eine Nachricht für das Overlay-Protokoll eintrifft.

Des Weiteren bieten die Erweiterungen verschiedene hilfreiche Methoden, wie etwa *sendMessage()* oder *presentNode()*.

Was muss beim Nutzen der Erweiterunge beachtet werden?
Jedes Modul, welches die Logik eines Overlayknotens enthält, muss von der Klasse `SimpleOverlayNode` erben. Die mit dem Modul verknüpfte C++ Klasse muss ebenso von `SimpleOverlayNode` erben und die Methoden *onInitialize()*, *onTimeout()* und *onMessageReceived()* implementieren. Jede Nachricht, die im Protokoll benutzt werden soll, muss von der Klasse `BaseOverlayMessage` aus der Datei „CommonMessages_m.h“ erben.

Wo finde ich weitere Informationen zu den Erweiterungen?
Die Erweiterungen sind nach der Einrichtung im Ordner „OverSim/src/overlay/simpleOverlay/“ zu finden. Sie sind komplett dokumentiert. Außerdem gibt es bereits implementierte Beispiele, aus denen die Benutzung der Erweiterungen ersichtlich werden sollte.

Gibt es eine Voranfertigung für ein neues Overlay-Netzwerk?
Ja, nach erfolgreicher Einrichtung findet sich ein Skelett im Ordner „OverSim/src/overlay/simpleOverlay_Skeleton“. Dieses kann als Einstieg für eine Overlay-Logik mit eigenen Nachrichtentypen verwendet werden.

Ich habe weitere Anmerkungen / Fragen zu den Erweiterungen!
Falls es sonst noch Fragen gibt, hilft eine Mail mit „VADs OverSim“ im Betreff an tillk@mail.upb.de.

3 Einrichtung

Hinweis: Alternativ zu der Einrichtung unten wird im Kurs auch eine virtuelle Maschine mit Ubuntu 12.04 LTS zur Verfügung gestellt, auf der OMNeT++ sowie OverSim und die Erweiterungen bereits komplett eingerichtet sind. Die Einrichtung der virtuellen

Maschine benötigt Oracle VirtualBox und wird am Ende dieser Sektion erklärt. Da die Einrichtung der virtuellen Maschine wesentlich komfortabler ist, sollte sie – falls möglich – bevorzugt werden.

3.1 Einrichtung ohne virtuelle Maschine

Ist es nicht möglich, die virtuelle Maschine zu verwenden, kann OMNeT mit OverSim auch manuell eingerichtet werden. Zunächst muss OMNeT++ eingerichtet werden. Aufbauend darauf kann dann OverSim in die Entwicklungsumgebung von OMNeT++ integriert werden. Darüber hinaus müssen noch Ordner importiert werden, die eine kleine Erweiterung von OverSim bereitstellen, um den Einstieg mit dem Framework zu Vereinfachen. Wir werden die Einrichtung hier in 5 kleineren Schritten erläutern:

1. Herunterladen der benötigten Dateien.
2. Einrichten der OMNeT++ IDE.
3. Einbinden von OverSim.
4. Einbinden der Erweiterungen.
5. Konfigurieren des Compilers.

3.1.1 Herunterladen der benötigten Dateien

Für eine funktionierende OverSim Umgebung brauchen wir mindestens die folgenden Dateien:

- OMNeT++ Version 4.2.2: Diese Version findet man auf der OMNeT++ Homepage im Bereich Older Versions.
- Das Inet Package in einer Version für OverSim. Dieses findet sich im OverSim Downloadbereich.
- OverSim in aktueller Version, zu finden ebenfalls im OverSim Downloadbereich.
- Die Erweiterungen zu diesem Kurs.

Hinweis: Für eine Installation unter Windows reichen die oben genannten Dateien aus. Für Linux / MacOS werden gegebenenfalls weitere Dateien benötigt. Hierzu findet man unter der OverSim Installationsanleitung für Linux oder der OverSim Installationsanleitung für MacOS nochmal eine ausführliche Anleitung.
Für Linux empfiehlt sich außerdem der Installation Guide am Ende dieses Dokuments. Man sollte hier besonders auf die benötigten Pakete und auf die richtigen Versionen dieser achten (Zu neue Versionen verursachen unter Umständen Probleme).

3.1.2 Einrichten der OMNeT++ IDE

Ist OMNeT++ 4.2.2 heruntergeladen und entpackt, muss es noch kompiliert werden. Hierzu findet man eine kurze Anleitung auf der Website von OverSim für Windows, Linux und MacOS. Bei Schwierigkeiten gibt es auch eine ausführliche Anleitung von OMNeT++ selbst am Ende dieses Dokuments in Sektion 8.

Hinweis: Für Linux empfiehlt es sich, dem Guide am Ende dieses Dokumentes genaustens zu folgen.

Hinweis: Die Einrichtung von OMNeT++ wurde erfolgreich auf Windows 10 und Ubuntu 12 getestet. Bei neueren Linux-Versionen sollte man darauf achten, nicht die neusten Versionen der benötigten Pakete zu laden.

3.1.3 Einbinden von OverSim

Beim ersten Start der OMNeT++ IDE legt man einen neuen Workspace an. Danach kann man über einen Rechtsklick im Workspace das Inet Package importieren. Dazu wählt man „Import -> existing projects into workspace“ und wählt den Ordners vom Inet Package aus. Auf die gleiche Art importiert man das OverSim Projekt.

3.1.4 Einbinden der Erweiterungen

Zu Vereinfachung des Arbeitens mit OverSim wurden ein paar Erweiterungen und Beispiele implementiert. Um diese Einzurichten gilt es, sie wie folgt zu kopieren:

- Die Datei „VADs_Example.ini“ in den Ordner „OverSim/simulations/“
- Die Ordner „simpleOverlay“, „simpleOverlay_Broadcast“, „simpleOverlay_Chain“, „simpleOverlay_Clique“, „simpleOverlay_Hello“ und „simpleOverlay_Skeleton“ in den Ordner „OverSim/src/overlay/“
- Der Ordner „application_doesnothing“ in den Ordner „OverSim/src/applications/“

3.1.5 Konfigurieren des Compilers

Standardmäßig compiled OMNeT++ unter der Richtlinie „Debug“. Dies kann aber zu schwer nachvollziehbaren Problemen in unserem Set-up führen (in denen manche Referenzen auf Klassen z.B. nicht berücksichtigt werden). Daher stellen wir den Compiler auf die Richtlinie „Release“ um. In der OMNeT++ IDE führen wir nun jeweils für die Projekte Inet und OverSim die folgenden Schritte aus:

1. Rechtsklick auf den Projektordner und Klick auf „Properties“
2. Auswählen von „C/C++ Build“
3. Neben „Configuration“ auf „Manage Configurations ...“ klicken

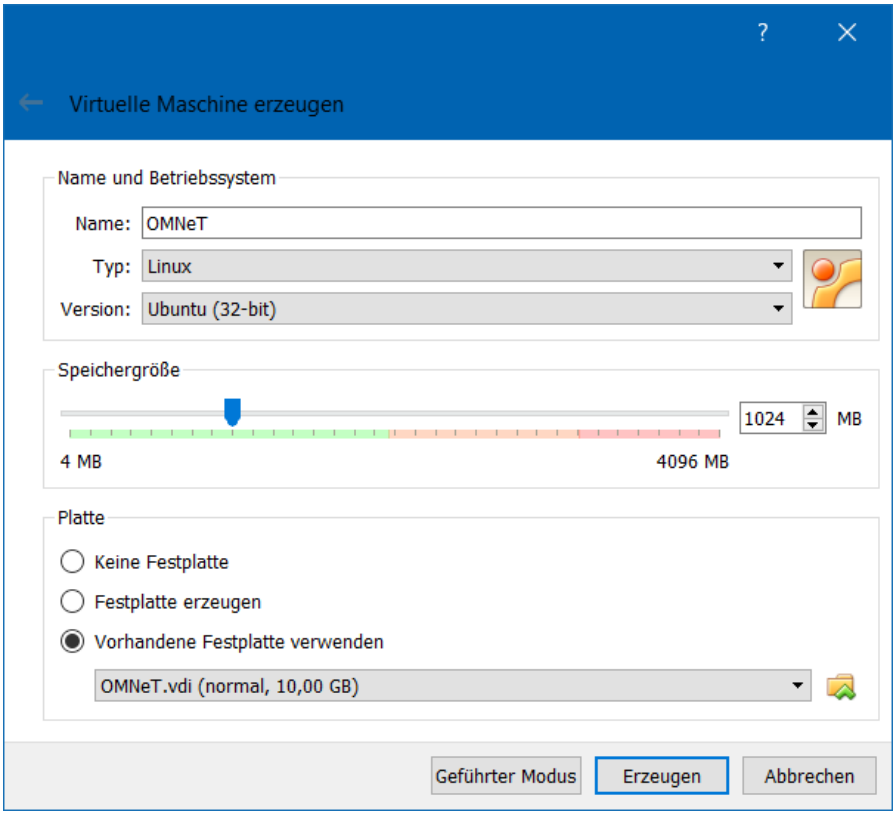
4. Anwählen von „gcc-release“
5. Klick auf „Set Active“. In der Tabelle sollte nun der entsprechende Eintrag auf „active“ stehen.
6. Klick auf „OK“
7. Klick auf „Apply“

Schließlich müssen wir nochmal alles compilen, indem wir aus der Menüleiste „Project -> Build All“ auswählen. Der Build Vorgang kann etwas dauern, aber danach sind wir bereit, OverSim zu benutzen.

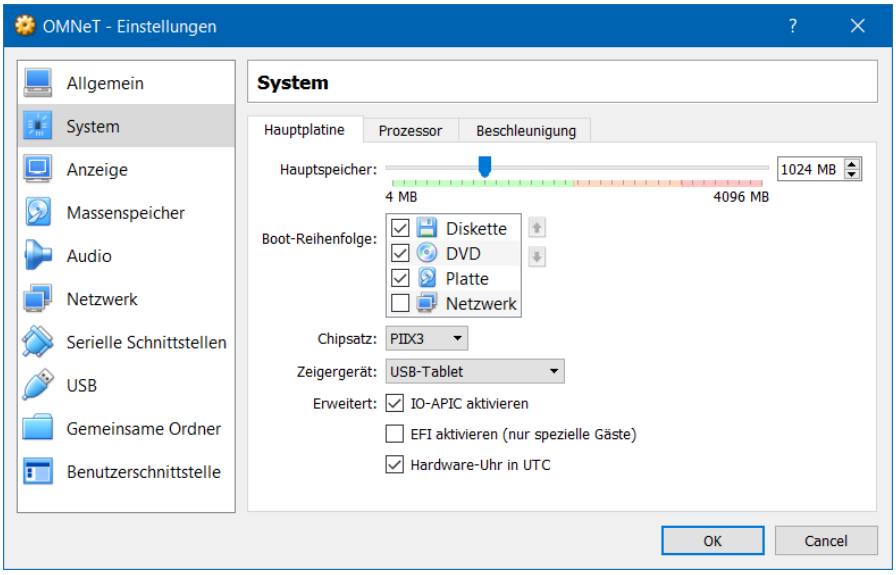
3.2 Einrichten als virtuelle Maschine

3.2.1 Einrichten

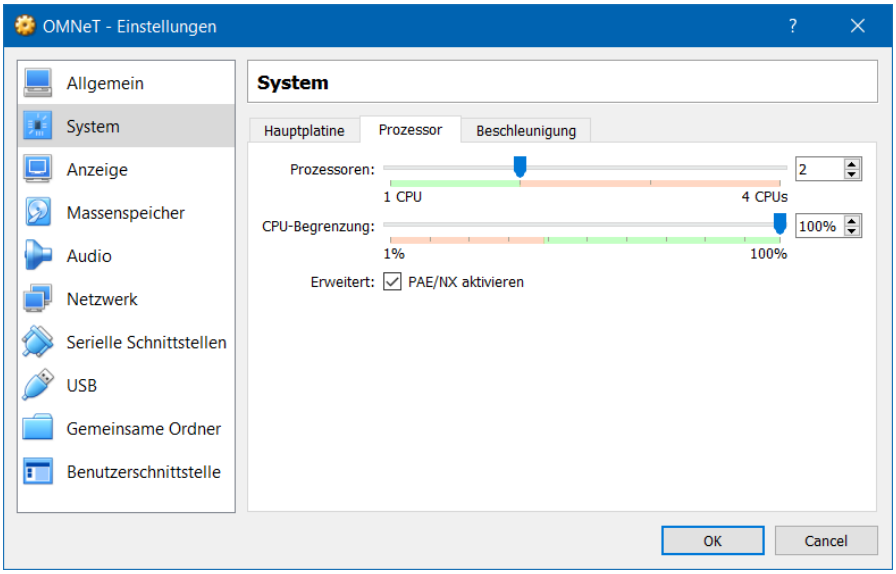
Für die virtuelle Maschine ist Oracle VirtualBox notwendig. Die Maschine wird als .vdi Datei angeboten. Ist VirtualBox installiert und gestartet, fügen wir mit „Neu“ eine neue virtuelle Maschine hinzu.

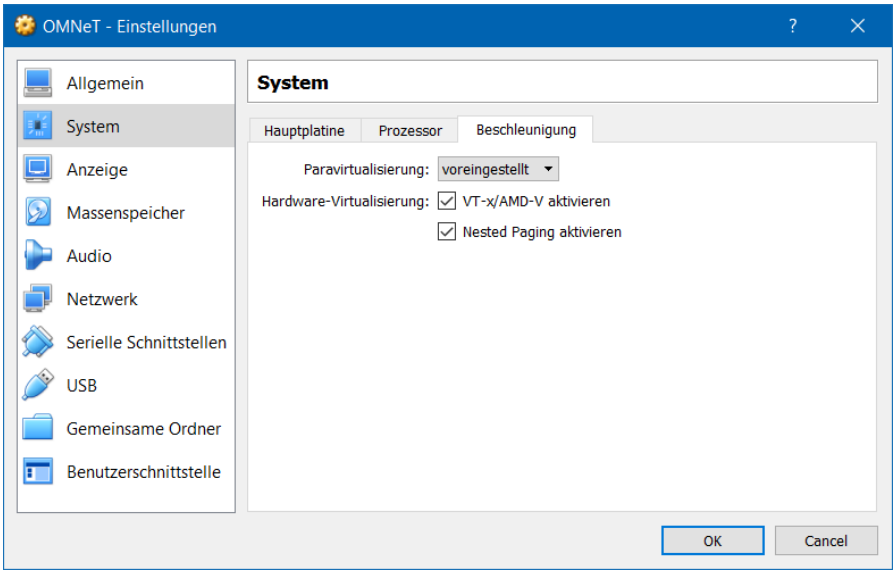


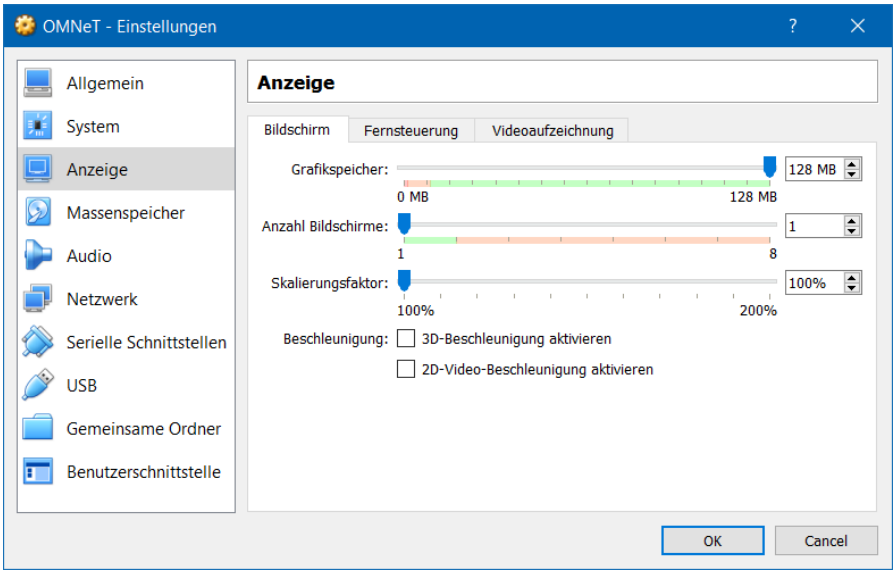
Sind die Eintellungen wie im Bild oben gesetzt, wird durch einen Klick auf „Erzeugen“ die virtuelle Maschine angelegt. Es gibt noch ein paar Kleinigkeiten, die wir einstellen müssen, indem wir die Maschine anwählen und auf „Ändern“ klicken.



Unter System ist wichtig, IO-Apic zu aktivieren. Dadurch können wir der Machine mehr als einen Prozessor zuweisen.







Ist die Maschine wie oben eingerichtet, können wir sie mit Klick auf „Starten“ booten und wir sehen den Ubuntu Desktop wie im Bild unten.

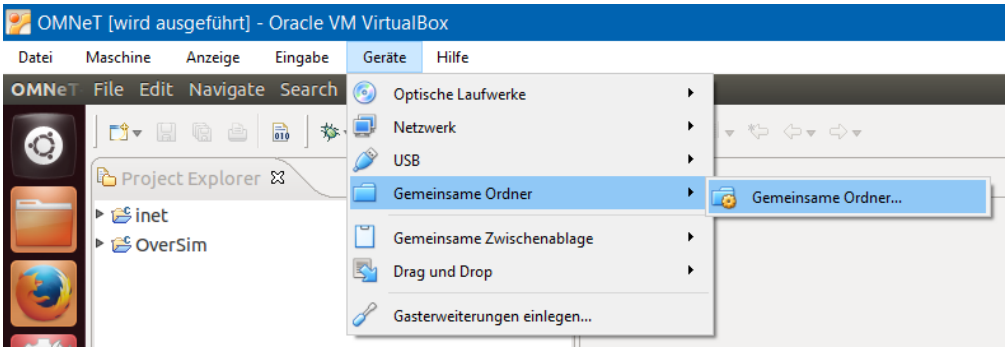
Hinweis: Die Zugangsdaten für das virtuelle Linuxsystem lauten:

- Benutzer: omnetpp
- Passwort: omnetpp

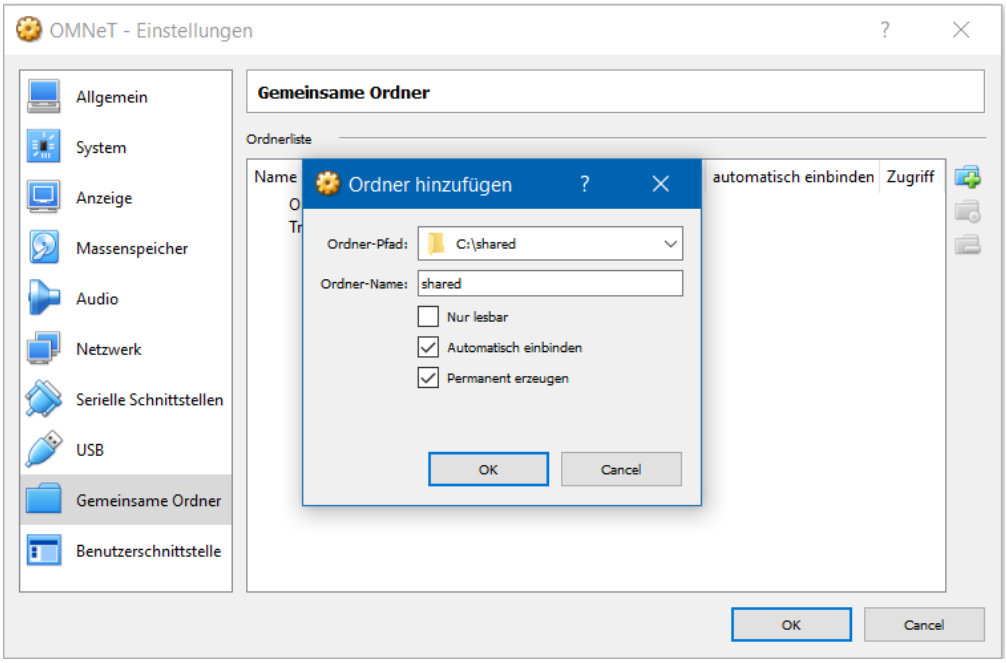
Hinweis: Unter Windows kann es zu Problemen kommen, wenn HyperV eingeschaltet ist. Hier kann es helfen, unter „Programme und Feaures -> Windows Funktionen“ HyperV zu deaktivieren und das System neuzustarten.

3.2.2 Erstellen eines geteilten Ordners

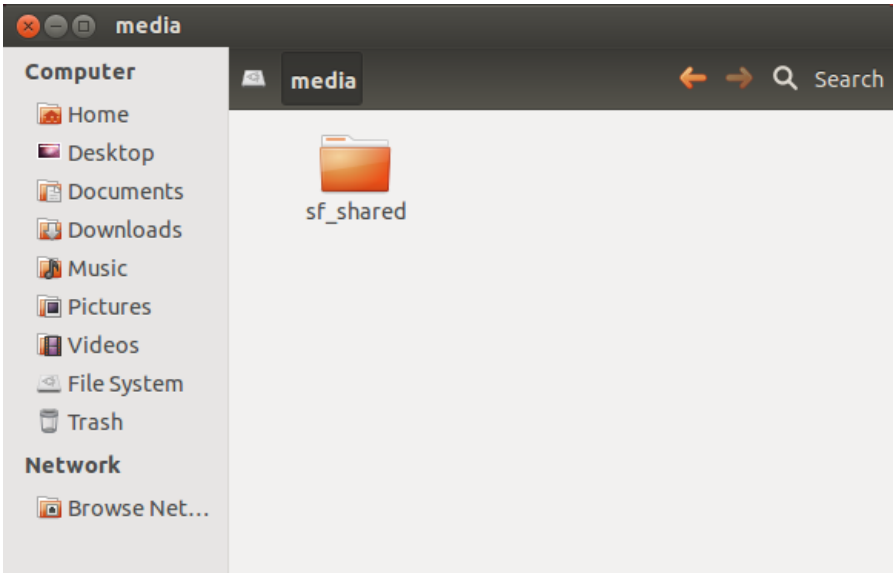
Mit VirtualBox ist es möglich einen Ordner zu erstellen, der zwischen dem Host System und der virtuellen Maschine geteilt wird. Aus der laufenden virtuellen Maschine wählen wir „Geräte -> Gemeinsame Ordner“.



Wir wählen einen Ordner im Hostsystem an und setzen die Haken wie im Bild unten.



Der Ordner erscheint nun in der virtuellen Maschine unter „File System/media/“.



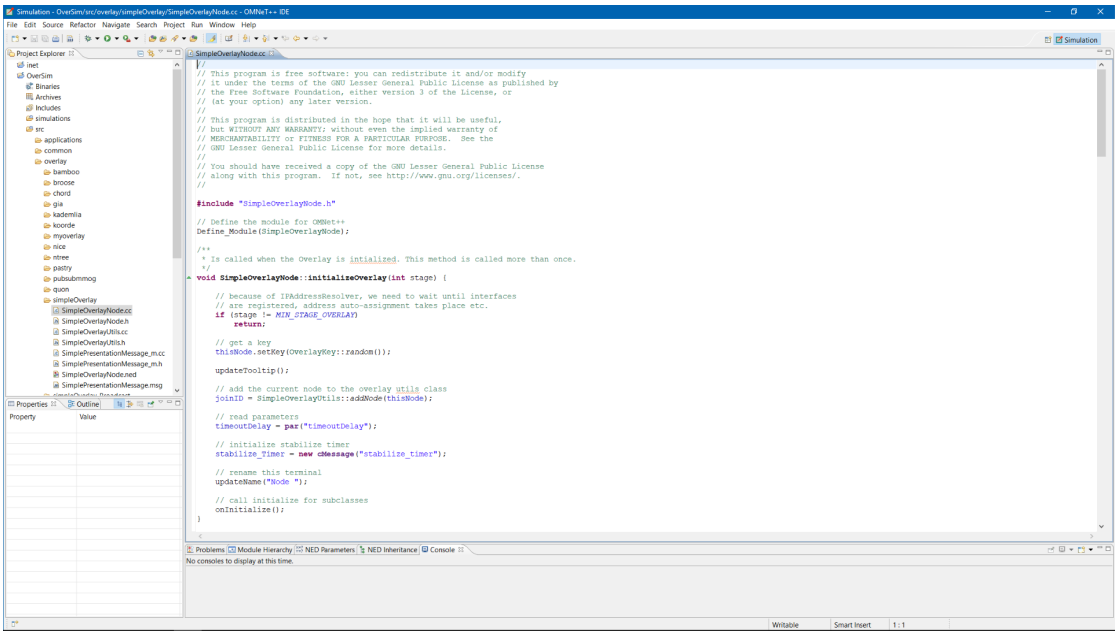
4 OMNeT++ IDE

4.1 Übersicht

Die OMNeT++ IDE basiert auf Eclipe und dürfte einem dementsprechend vertraut vorkommen. Unter Windows startet man die Datei „mingwenc.cmd“ im Ordner von OMNeT++ und gibt den Befehl *omnetpp* ein. Unter Linux öffnet man ein Terminal im OMNeT++ Ordner und nutzt die Befehle . *setenv* und *omnetpp*, um die IDE zu öffnen.

Hinweis: Unter Linux kann mit Hilfe von . *setenv* und *make install-desktop-icon* ein Desktop Icon für die IDE erstellt werden. Diese Verlinkung existiert bereits in der virtuellen Maschine.

Nach dem Start sieht man die IDE wie unten vor sich:



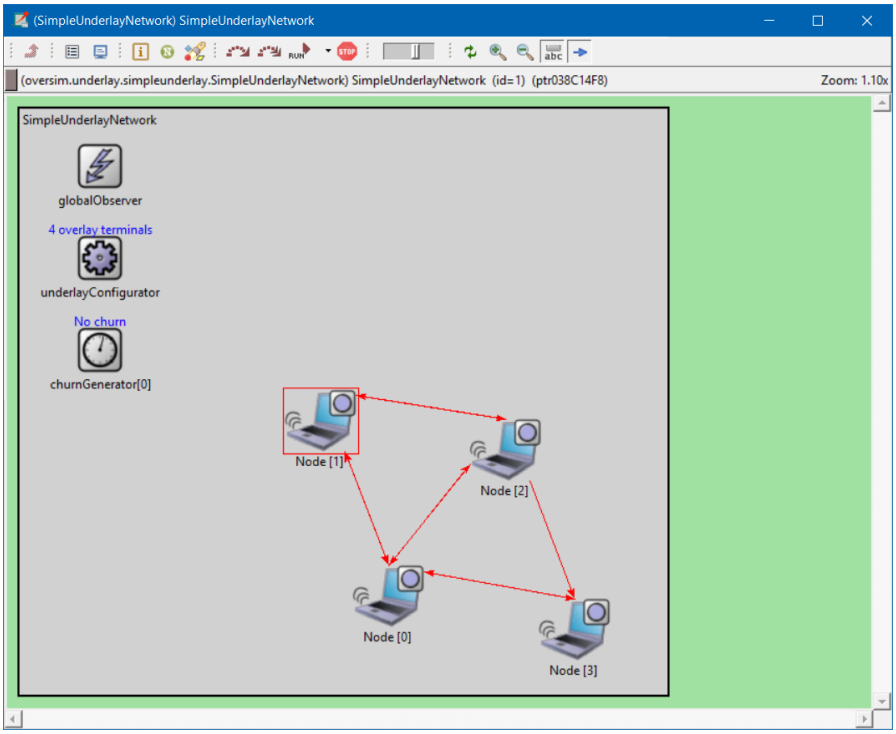
Wie in Eclipse sieht man links alle Projekte, in der Mitte geöffnete Editoren und unten eine Ausgabe und weitere Informationen zum Programm.

Hinweis: Wenn bestimmte Ordner / Dateien im OverSim Projekt geöffnet sind, kann es sein, dass die IDE Fehler anzeigt und beim Build-Vorgang anmerkt, dass solche existieren. Grundsätzlich gilt: Alle relevanten Fehler werden während des Build-Vorgangs in der Console in rot angemerkt. Sind hier keine Fehler angemerkt, kann die Simulation in der Regel problemlos gestartet werden.

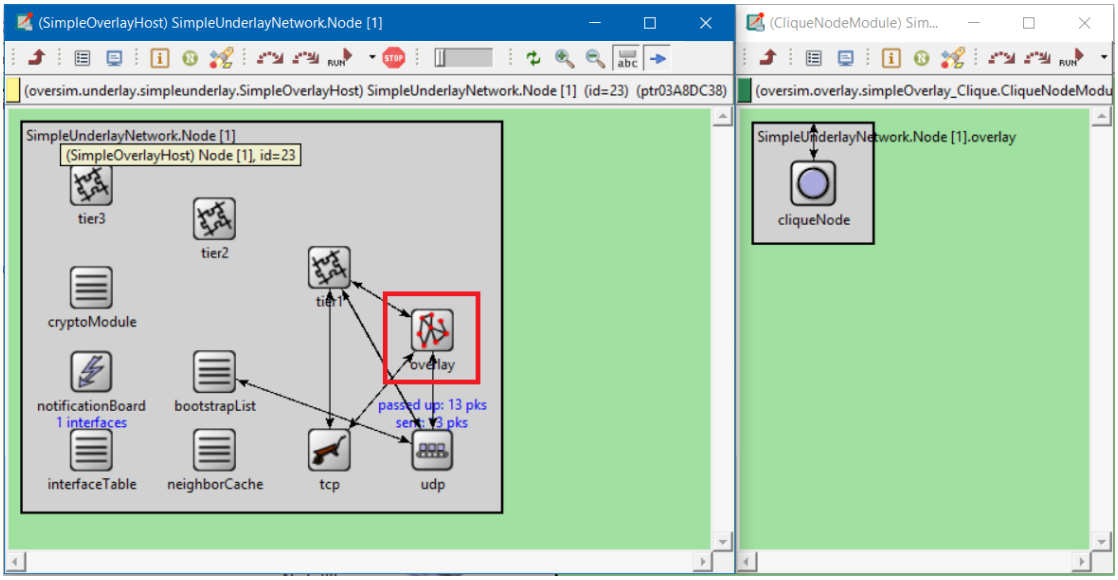
Hinweis: Wurden eigene Message Klassen implementiert, muss mindestens zweimal compiled werden. Beim ersten Compilen werden aus der .msg Definition noch C++ Dateien generiert.

4.2 Oberfläche während der Simulation

Während einer Simulation in OMNet++ öffnen sich zwei Fenster. Eines davon ist eine graphische Darstellung des Overlay Netzwerkes.

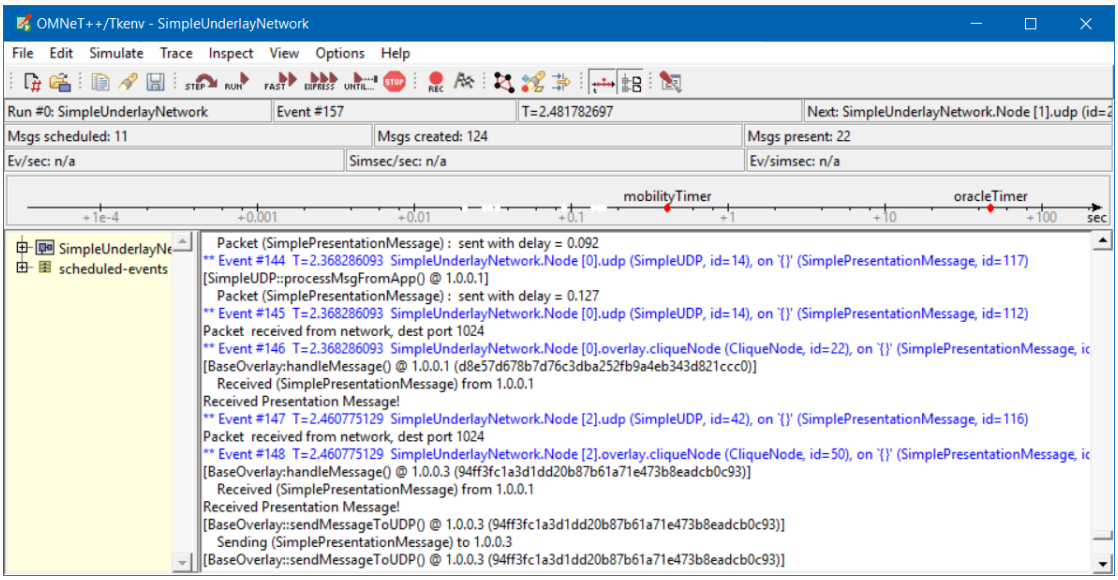


In der oberen Leiste können wir die Simulation kontrollieren. Hilfreicht ist hier der *Re-Layout* Button, mit dem die graphischen Objekte automatisch neu angeordnet werden. Während der Simulation kann es nämlich durchaus vorkommen, dass Objekte übereinander dargestellt werden. Unsere Overlay-Teilnehmer werden durch die Laptop-Icons dargestellt. Das Hinzufügen neuer Teilnehmner übernimmt der *Churn Generator*. Mit einem Rechtsklick auf ein Modul können wir uns sowohl den Output des Moduls als auch alle seine aktiven Objekte ansehen. Das aktuell aktive Modul wird durch einen roten Rahmen markiert (Node[1] im Bild oben).



Mit einem Rechtsklick auf das Modul kann man es graphisch inspizieren (siehe Bild oben). Hier sieht man gut die Struktur von OverSim. Das in rot markierte Modul (*overlay*) bettet unsere Implementierung ein. Die anderen Module werden von OverSim gestellt. Inspiziert man dieses Modul weiter, sieht man genau das von uns implementierte Modul *cliqueNode* (rechts im Bild).

Hinweis: Zum Nachverfolgen von Objekten eines Moduls kann im Code das *Watch(object)* Statement benutzt werden.



Das zweite Fenster bietet uns einen vollständigen Überblick über alle geloggten Na-

chrichten sowie eine Zeitleiste. Auch hier können wir die Simulation steuern. Außerdem erlaubt uns dieses Fenster das Nutzen eine Express Modus in dem jede graphische Ausgabe unterbunden wird, um die Simulation mit maximaler Performance durchzuführen.

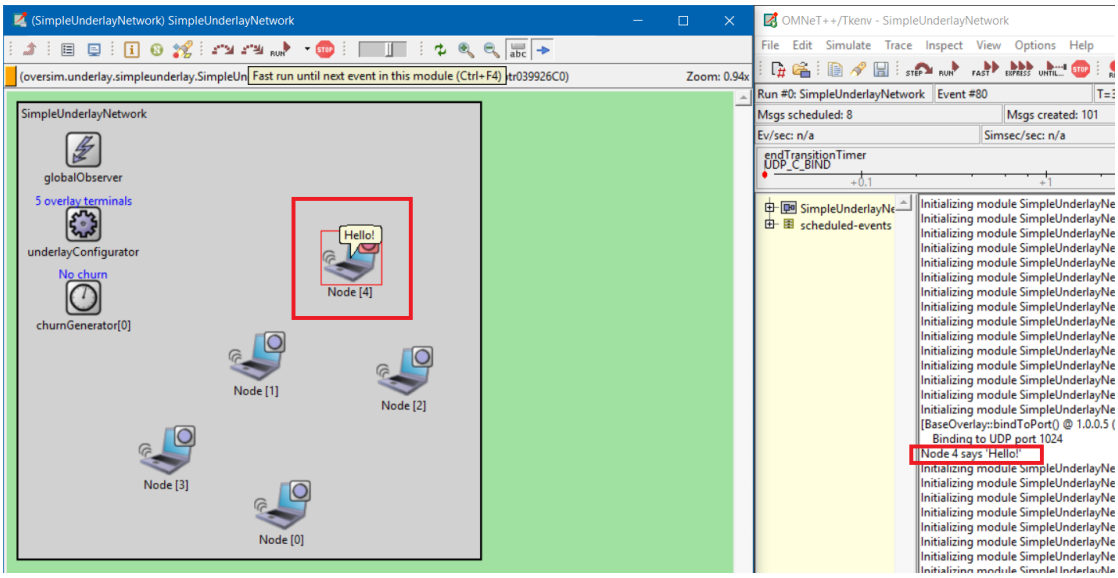
Hinweis: Zum Ausgeben von Text kann im Code das *EV « text* Statement benutzt werden.

5 Einführungsbeispiele

Zum leichteren Verständnis wurden einige Beispiele implementiert, die die Verwendung von OMNeT++ und OverSim verdeutlichen sollen. Zum Starten der Beispiele gilt es, in der OMNet++ IDE mit einem Rechtsklick „Run as -> OMNet++ Simulation “ anzuwählen. Im Folgenden wird man nach der Konfiguration gefragt, die man ausführen möchte. Hier wählen wir die Datei „VADs_Example.ini“. Beim Starten können wir nun in einem Dropdown Menü eines der unten aufgeführten Beispiele zum Simulieren auswählen.

5.1 Hello

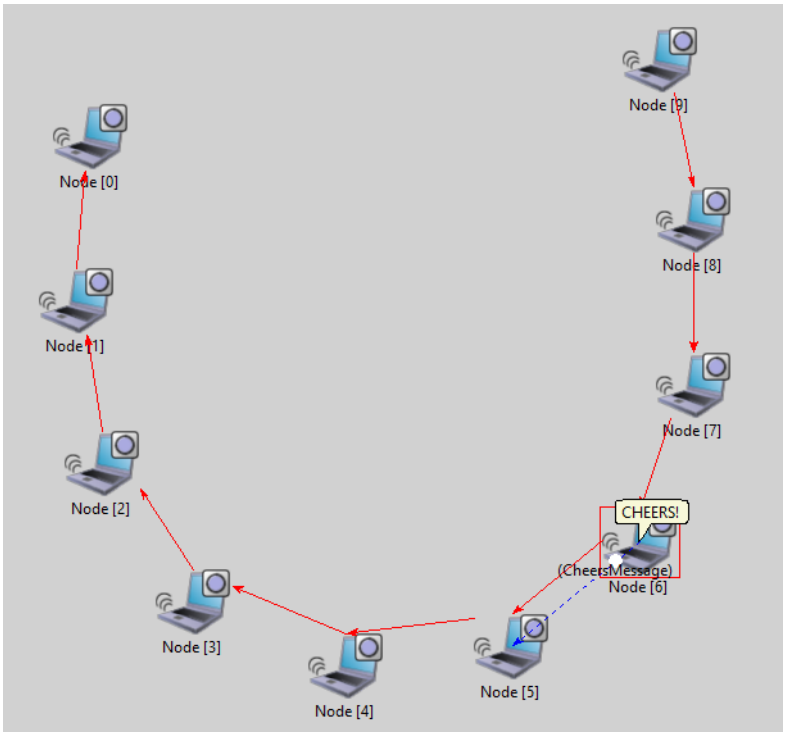
In diesem Beispiel wir jeder Knoten beim Eintreten ins Netzwerk eine Sprechblase erzeugen, in der er sich meldet. Außerdem können wir das „Hello“ eines Knotens im Log sehen. Zu finden ist das Beispiel im Ordner „OverSim/src/overlay/simpleOverlay_Hello/“.



5.2 Chain

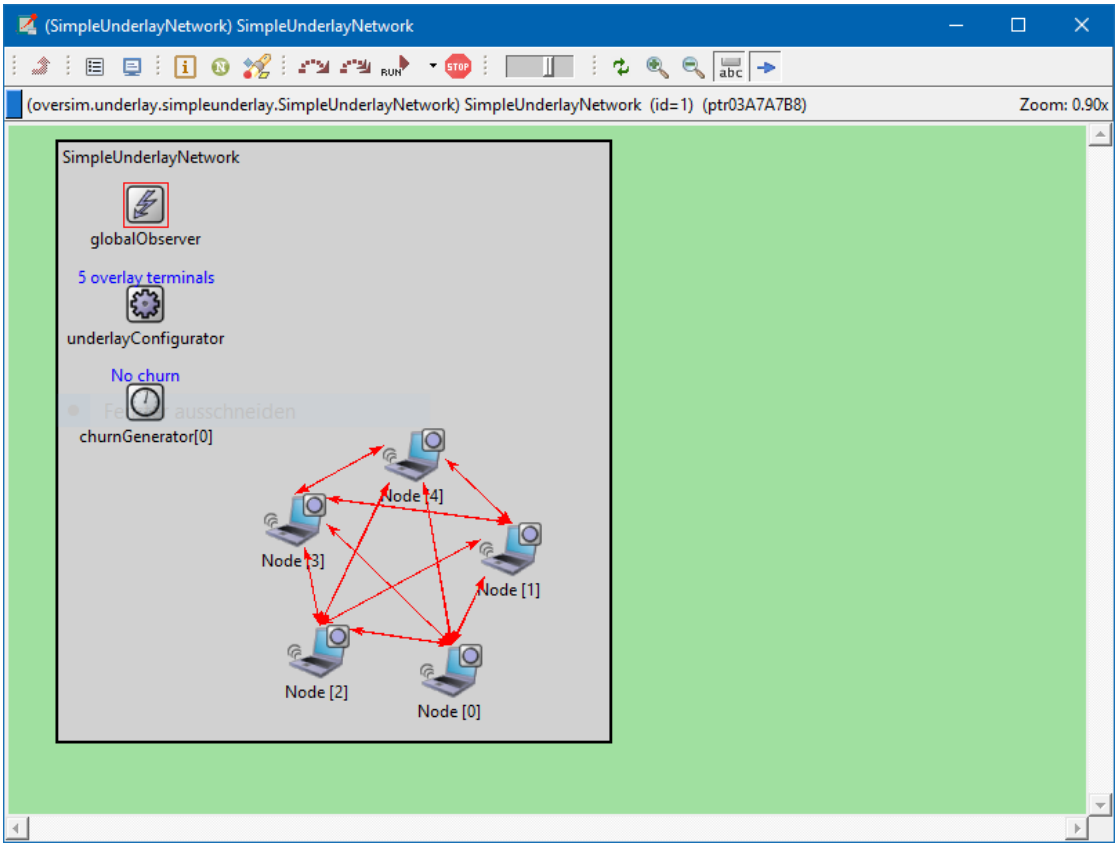
In diesemn Beispiel bilden unsere Prozesse eine Kette. Der letzte Knoten, der dem Netzwerk beitritt, wird der Kopf der Kette. Er sendet in regelmäßigen Abständen eine

Nachricht los, die einmal entlang der Kette gereicht wird. Das Beispiel befindet sich nach der Einrichtung im Ordner „OverSim/src/overlay/simpleOverlay_Chain/“.



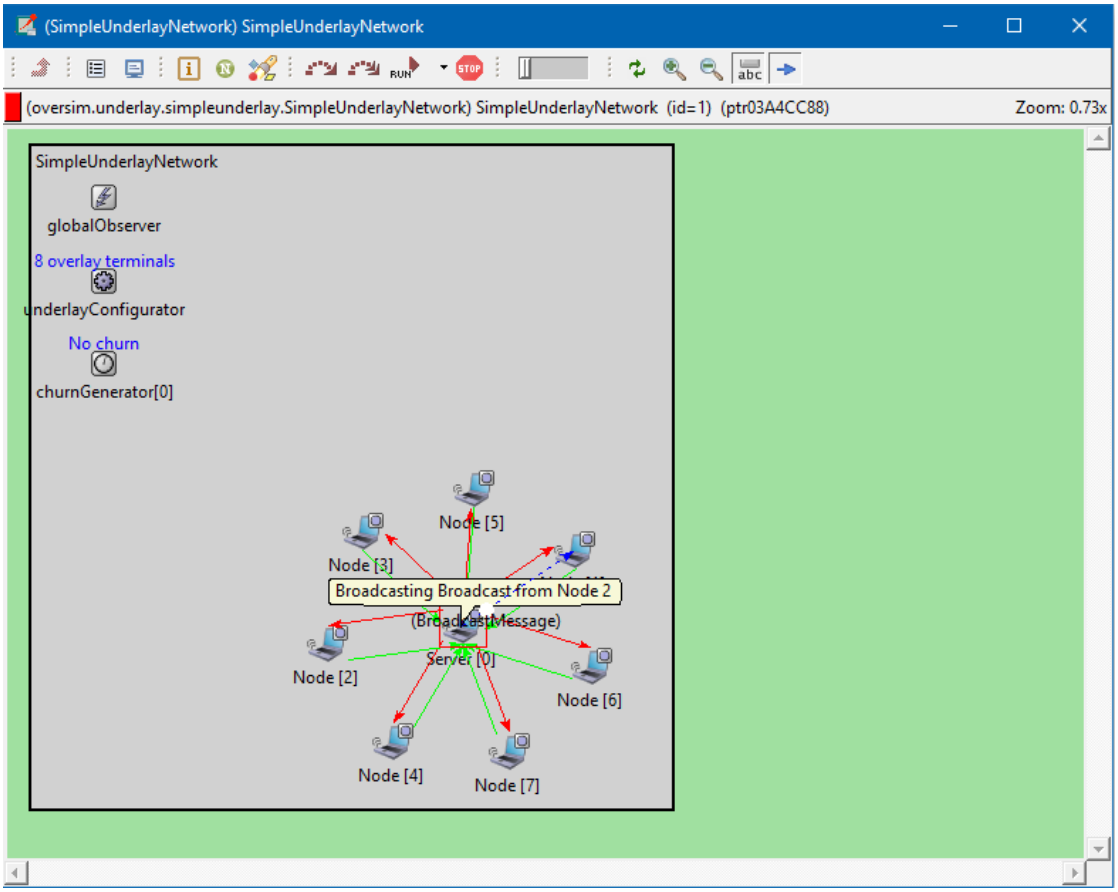
5.3 Clique

In diesem Beispiel bilden die simulierten Knoten eine Clique. Dafür stellt sich ein eintretender Knoten allen anderen vor und jeder Knoten verteilt einen neuen Knoten an alle seine Nachbarn. Die Implementierung liegt im Ordner „OverSim/src/overlay/simpleOverlay_Clique/“



5.4 Broadcast

Mit diesem Beispiel wird ein Broadcast simuliert. Der erste Teilnehmer erhält die Rolle des Servers. Alle weiteren Teilnehmer sind Clients. Ein Client sendet mit einer geringen Wahrscheinlichkeit eine Nachricht an den Server, die dann als Broadcast an alle Clients verteilt wird. Dieses Beispiel befindet sich im Ordner „OverSim/src/overlay/simpleOverlay_Broadcast/“

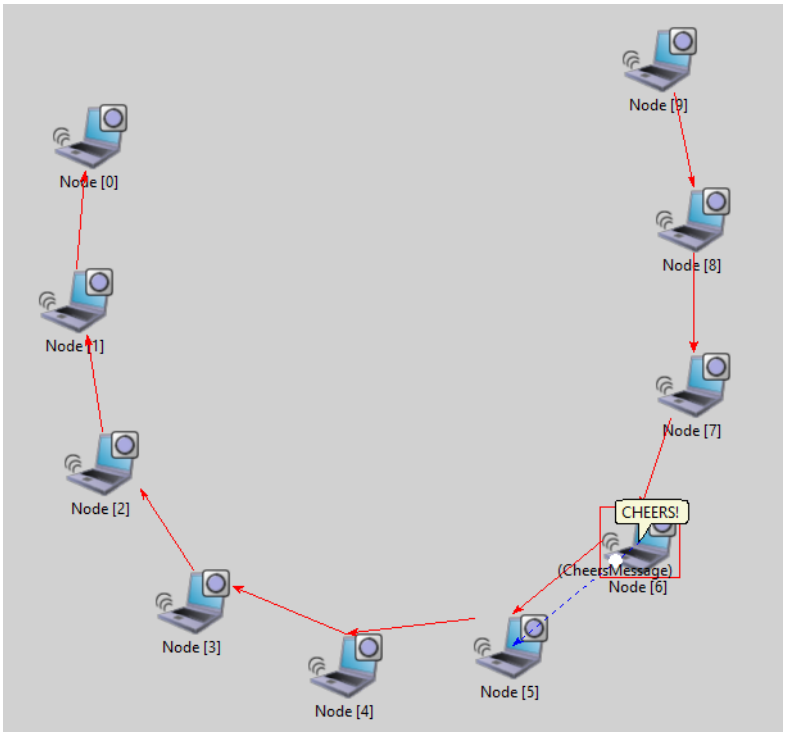


6 Beispiel: Einfache Kette

In der folgenden Sektion wird am Beispiel der bereits implementierten einfachen Kette gezeigt, wie eine Implementierung in OverSim funktioniert.

Hinweis: Die Erläuterung dieses Beispiels erfolgt aufbauend auf den oben vorgestellten Erweiterungen.

Die Implementierung, die im folgenden beschrieben wird, findet sich nach vollständiger Einrichtung im Ordner „OverSim/src/overlay/simpleOverlay_Chain/“.
Ziel der Implementierung ist es, Knoten in einer Kette anzuordnen. Der Anfang der Kette wird in regelmäßigen Abständen eine Nachricht lossenden. Jeder Knoten, der diese Nachricht empfängt, zeigt auf der GUI „Cheers!“ an und sendet die Nachricht danach an seinen Nachfolger weiter. Am Ende wird das dann wie im Bild unten aussehen:



OverSim fügt bei der Simulation die Overlay-Knoten nach und nach dem Netzwerk hinzu. Einem eintretenden Knoten wird dabei ein bereits existenter Knoten präsentiert. Das führt dazu, dass wir unsere Kette während der Simulation quasi rückwärts aufbauen. Der Kopf der Kette wird am Ende der letzte Knoten sein, der in das System eingetreten ist.

6.1 Anlegen des neuen Netzwerkes

OverSim bietet bereits eine Umgebung, in der bei der Simulation der von uns definierte Overlay-Knoten und seine Logik eingebettet werden. Daher gilt es für uns, diese Einbettung in den nachher simulierten Knoten zu definieren. Mit einem Rechtsklick auf unseren Ordner und dem Befehl „New->Simple Module“ erzeugen wir eine Datei „ChainNode.ned“, in der wir diese Einbettung festlegen. `ChainNode` heißt nachher das OMNeT++ Modul, welches sämtliche Logik unseres Overlay beinhaltet. Standardmäßig erbt das Modul von `cSimpleModule`. Wir lassen `ChainNode` allerdings von `SimpleOverlayNode` erben, denn dieses Modul bietet alle für uns relevanten Methoden an. Der Befehl *@class(ChainNode)* verknüpft unser Modul mit einer C++ Klasse, in der wir später die Logik implementieren. Zusätzlich brauchen wir noch einen Parameter *nodeNumber*, damit ein Knoten später darauf schließen kann, dass er der letzte Knoten der Chain und damit der Kopf ist.

```
//
// A ChainNode is a SimpleOverlayNode
//
simple ChainNode extends SimpleOverlayNode
{
    // Connect this model with our c++ class
    parameters:

        @class(ChainNode); //< connects our module with our c++ class for the logic

        int nodeNumber; //< Parameter for the total number of nodes
}
```

Nun müssen wir unser Overlay-Modul noch mit dem von OverSim erzeugten Simulationsknoten verbinden. Dazu legen wir unterhalb des `ChainNode` Moduls ein Modul `ChainNodeModule` fest. Es integriert `IOverlay` und hat verschiedene Gates zu den anderen Layern des Simulationsknotens (z.B. dem UDP Layer). Außerdem hat es genau ein Objekt vom Typ `ChainNode` und definiert Verbindungen vom ChainNode zu den anderen Komponenten des Simulationsknotens. Mit der Definition dieses Moduls haben wir dafür gesorgt, dass nachher jeder simulierte Knoten eine Instanz von `ChainNode` hat, die OverSim über die standardisierten Gates vom Interface `IOverlay` ansprechen kann.

```
//
// This module embeds our Overlay node in the Simulation
//
module ChainNodeModule like IOverlay
{
    // gates to the other layers of a node
    gates:
        input udpIn;    // gate from the UDP layer
        output udpOut;    // gate to the UDP layer
        input tcpIn;    // gate from the TCP layer
        output tcpOut;    // gate to the TCP layer
        input appIn;    // gate from the application
        output appOut;    // gate to the application

    // the module that holds the overlay protocol
    submodules:
        chainNode: ChainNode;

    // connects the overlay module with the other layers
    connections allowunconnected:
        udpIn --> chainNode.udpIn;
        udpOut <-- chainNode.udpOut;
        appIn --> chainNode.appIn;
        appOut <-- chainNode.appOut;
}
```

6.2 Definieren der Overlay-Logik

Da wir nun dafür gesorgt haben, dass unser Overlayknoten durch OverSim ordnungsgemäß simuliert werden kann, können wir nun die Logik unseres Protokolls in den automatisch erzeugten Dateien „ChainNode.h“ und „ChainNode.cc“ implementieren.

Schauen wir zunächst in die Header Datei „ChainNode.h“: Auch hier muss unser `ChainNode`

wieder von `SimpleOverlayNode` erben. Des Weiteren brauchen wir ein paar Variablen; einen *successor*, der den Nachfolger des aktuellen Knoten darstellt, und ein Flag *head*, welches angibt, ob der aktuelle Knoten den Anfang der Kette darstellen wird. Die Entscheidung eines Knotens, ob er der Anfang der Kette ist, treffen wir später in der Initialisierung.

Hinweis: Informationen über Overlay-Knoten werden in Objekten vom Typ `NodeHandle` gespeichert. Jeder Knoten hat auch eine Referenz dieses Typs auf sich selbst im Feld *thisNode*.

```
/**
 * This class represents a simple overlay node that is used for our chain example.
 */
class ChainNode: public SimpleOverlayNode {

private:

    NodeHandle successor; //< the successor of this ChainNode

    bool head; //< determines if the current node is the head of the chain
```

Die nachfolgenden *public* Methoden *onInitialize()*, *onTimeout()* und *onMessageReceived()* überschreiben jeweils Methoden der Superklasse. Sie werden dadurch alle automatisch zum entsprechenden Zeitpunkt aufgerufen. *updateTooltip()* dient dazu, die GUI zu manipulieren um z.B. Pfeile zu anderen Knoten zu zeichnen. Diese Methode muss nicht überschrieben werden, wir implementieren sie aber trotzdem neu, damit unser Netzwerk beim Simulieren gut aussieht.

```
public:

    /**
     * Gets called when the node is initialized.
     */
    void onInitialize();

    /**
     * Gets called continuously during simulation.
     */
    void onTimeout();

    /**
     * Gets called when a message for the Overlay Protocol is received.
     */
    void onMessageReceived(BaseOverlayMessage* msg);

    /**
     * Updates the GUI.
     */
    void updateTooltip();
};
```

Nun geht es an die eigentliche Implementierung in „ChainNode.cc“: Zunächst brauchen wir unbedingt den Befehl *Define_Module(ChainNode).* Dieser weist OMNeT nochmal an, dass nun die Implementierung des definierten Moduls `ChainNode` stattfindet. Danach implementieren wir alle im Header angegebenen Methoden.
Zunächst legen wir fest, was bei der Initialisierung eines Knotens geschieht. Wir legen den *successor* auf einen unspezifizierten Knoten fest. Das Statement *WATCH(sucessor)* sorgt dafür, dass wir in der GUI später den Wert des Feldes ansehen können. Danach müssen wir festlegen, ob der aktuelle Knoten der letzte – und damit der Anfang der Kette – ist. Dazu lesen wir mit *par("nodeNumber")* den im Modul `ChainNode` festgelegten Parameter. Die *joinID* wird fortlaufend einem neuen Knoten beim Eintritt ins Netzwerk zugewiesen. Diese Nummerierung startet bei 0. Wenn der aktuelle Knoten Anfang der Kette ist, bewirkt der Befehl *yell()*, dass in der GUI zu diesem Zeipunkt eine Sprechblase am Modul erscheint. Der erste Knoten hat keinen *successor*. Jedem anderen Knoten wird im Folgenden ein Verweis auf den vor ihm eingetretenen Knoten gegeben. `SimpleOverlayUtils` stellt hierbei eine Erweiterung von OverSim dar, die uns ermöglicht, einen `NodeHandle` von jedem Knoten nur auf Basis seiner *joinID* zu bekommen. Mit *EV « text* wird ein String *text* als Nachricht in der Konsole während der Simulation ausgegeben. Am Ende der Methode rufen wir noch *updateTooltip()* auf, um die GUI neu zu zeichnen, denn nun besitzt unser Knoten unter Umständen einen Verweis auf einen anderen Knoten.

Hinweis: OverSim gibt einem Overlay-Knoten unter der Haube einen zufälligen Wert als Schlüssel. Dieser befindet sich in *thisNode.getKey().* In OverSim wird ein Knoten nicht nur durch seinen Overlay-Schlüssel identifiziert, es muss auch seine IP Adresse bekannt sein. Die *joinID* wurde als Erweiterung implementiert, damit man leichter auf andere Knoten zugreifen kann. Dieses Verfahren ist unrealistischer als das von OverSim, vereinfacht aber die Handhabung, wie in diesem Beispiel zu sehen ist.

```
/**
 * Gets called when the node is initialized.
 */
void ChainNode::onInitialize() {
    // set the child to an empty node
    successor = NodeHandle::UNSPECIFIED_NODE;

    // We want to keep track of the successor in the GUI
    WATCH(successor);

    // are we the head of the chain?
    head = false;

    if (par("nodeNumber").doubleValue() == joinID + 1) {
        // we are the head
        yell("I am the head!");
        head = true;
    }

    if (joinID > 0) {
        // we are not the first node!

        // get the successor
        successor = *(SimpleOverlayUtils::getNode(joinID - 1));

        // yell the good news!
        yell("I found a successor!");

        EV << "My successor is node "
                << SimpleOverlayUtils::getJoinID(successor) << endl;
    }

    // update the GUI
    updateTooltip();
}
```

In der Methode *onTimeout()* soll nun nur der Anfang der Kette aktiv werden und eine Nachricht an seinen Nachfolger senden. Dazu senden wir einen Pointer auf eine `CheersMessage` mit der bereits in `SimpleOverlayNode` implementierten Methode *sendMessage()*. Die Implementierung der Nachricht werden wir im nächsten Abschnitt vornehmen.

```
/**
 * Gets called continuously during simulation.
 */
void ChainNode::onTimeout() {
    // If we are the head of the chain, we will send  a "cheers"
    if (head) {
        // we are the head node
        yell("Send CHEERS");
        CheersMessage* msg = new CheersMessage();
        sendMessage(msg, successor);
    }
}
```

Erreicht eine Nachricht einen Knoten, wird die Methode *onMessageReceived()* aktiv. Jede Nachricht, die das Overlay Protokoll betrifft, ist eine Subklasse von `BaseOverlayMessage`.

Mit Hilfe von *dynamic_cast* können wir testen, welcher Art die Nachricht ist. `SimplePresentationMessage` ist ein einfacher Nachrichtentyp, der einen Overlay-Knoten vorstellen kann. Da diese Funktionalität nicht in diesem Beispiel vorkommt, ist für uns nur der zweite Fall interessant, in dem eine `CheersMessage` empfangen wird. Erhalten wir eine solche, zeigen wir „CHEERS!“ an. Sofern der aktuelle Knoten nicht das Ende der Kette ist (und damit der *successor unspecified* ist), senden wir die Nachricht einfach weiter.

```
/**
 * Gets called when a message for the Overlay Protocol is received.
 */
void ChainNode::onMessageReceived(BaseOverlayMessage* msg) {
    // determine message type
    if (SimplePresentationMessage* message=dynamic_cast<SimplePresentationMessage*>(msg)) {
        // we received a presentation message
        EV << "Received Message!";
        // This will be some node of the network that has been introduced to us as anchor
        // We don't need it in this example
        // delete the message
        delete msg;

    } else if (CheersMessage* message=dynamic_cast<CheersMessage*>(msg)) {
        // we received a cheers message! Join in!
        yell("CHEERS!");
        // if we have a successor, forward the message
        if (!successor.isUnspecified()) {
            // Debug output
            EV << "Node " << SimpleOverlayUtils::getJoinID(thisNode)
                    << " is forwarding Cheer Message" << endl;
            sendMessage(msg, successor);
        } else {
            // we are at the end of the chain :( delete the message
            delete msg;
        }
    }
}
```

Es fehlt noch die Methode *updateTooltip()*, die die GUI neu zeichnet und damit unser Beispiel beim Simulieren wesentlich anschaulicher macht. Sie besteht aus zwei wesentlichen Teilen. Zunächst ändern wir den Text, der bei Mouseover über unser Modul angezeigt wird, sodass wir auch die *joinID* unseres *successor* sehen. Außerdem zeichnen wir einen Pfeil vom aktuellem Knoten zu seinem *successor*.

```
/**
 * Updates the GUI.
 */
void ChainNode::updateTooltip() {
    // is this module displayed?
    if (ev.isGUI()) {
        // show this node in the tooltip

        std::stringstream ttString;

        if (!successor.isUnspecified()) {
            ttString << "This: " << joinID << endl << "Succ: "
                 << SimpleOverlayUtils::getJoinID(successor);
        } else {
            ttString << "This: " << joinID;
        }

        updateMouseOver(ttString.str().c_str());

        // draw a red arrow to our successor
        showOverlayNeighborArrow(successor, true, "m=m,50,0,50,0;ls=red,1");
    }
}
```

6.3 Definieren von neuen Nachrichten

Ein Teilnehmer unserer Kette reagiert in seiner *onMessageReceived()* Methode nur, wenn er eine `CheersMessage` erhalten hat. Nun gilt es, diesen Nachrichtentypen noch zu implementieren. Wir legen dazu eine neue „Message Definition (msg)“ an, die wir `CheersMessage` nennen.

```
// c++ code to include BaseOverlayMessage (is defined in CommonMessages_m.h)
cplusplus {{
#include <CommonMessages_m.h>
}}

// the Superclass
class BaseOverlayMessage;

//
// A Message that indicates the receiving node should cheer.
//
packet CheersMessage extends BaseOverlayMessage {
    // does not contain anything
}
```

Hinweis: Beim ersten Compilen wird für eine „Message.msg“ Definition automatisch zwei C++ Dateien „Message_m.h“ und „Message_m.cc“ erzeugt. Die komplette Nachricht sollte aber nur in der .msg Datei definiert werden (unter Umständen muss also mehrmals compiled werden).

Jeder Nachrichtentyp ist vom Typ `packet` und muss von `BaseOverlayMessage` erben, damit die Nachricht reibungslos von OverSim an unseren Overlay-Knoten gericht wird.

Dieser Typ ist definiert in „CommonMessages_m.h“. Im oberen Teil der Datei geben wir notwendige Includes für die C++ Definition der Nachricht an. Außedem geben wir eine Vorwärtsdefinition für die Superklasse an. Unsere `CheersMessage` enthält keine Informationen, daher müssen wir in der eigentlichen Nachricht nichts mehr implementieren.

6.4 Konfigurationen anlegen

Unser Overlay ist nun soweit fertig implementiert. Jetzt müssen wir OverSim noch klar machen, wie es unser Netzwerk zu simulieren hat. Dazu begeben wir uns in den Ordner „OverSim/simulations/“. Hier legen wir die Datei „VADs_examples.ini“ an. In dieser Datei legen wir fest, welche Parameter unser Overlay haben soll. Zunächst aber legen wir eine neue Konfiguration mit der Zeile *[Config Chain_Example]* an. Für unser Overlay müssen wir die folgenden Parameter festlegen:

- ***.overlayType* : Hier verweisen wir auf unser ChainNodeModule. OverSim wir daraufhin Overlay-Knoten erzeugen, die das von uns definierte Modul enthalten.
- ***.targetOverlayTerminalNum* : Die maximale Anzahl an Teilnehmern im Overlay Netzwerk.
- ***.nodeNumber* : Muss mit ***.targetOverlayTerminalNum* übereinstimmen. Wir brauchten diesen Parameter, damit ein Knoten weiß, ob er der letzte – und damit der Anfang der Kette – ist.
- ***.timeoutDelay* : Der Delay zwischen zwei Aufrufen von *onTimeout()*. Wir setzen ihn hier auf 40s. Dieser Parameter muss nicht zwangsläufig gesetzt werden, denn er hat einen Standardwert von 20s. Wir wollen die Simulation hier allerdings etwas verlangsamen.
- ***.tier1Type* : OverSim benötigt auch die Definition einer Anwendung, die Anfragen an das Overlay Netzwerk generiert. Der Einfachheit halber wurde bereits eine Anwendung implementiert, die nichts tut. Dies ist nützlich, wenn man einfach nur das Overlay Protokoll nachvollziehen will.

Zusätzlich benutzen wir noch „include ./default.ini“ für einige praktische Standardwerte für die Module von OverSim.

```
[Config Chain_Example]
description = Chain Example
**.overlayType = "oversim.overlay.simpleOverlay_Chain.ChainNodeModule"
**.targetOverlayTerminalNum = 10
**.nodeNumber = 10
**.timeoutDelay = 40s
**.tier1Type = "oversim.applications.application_doesnothing.DoesNothingAppModules"

include ./default.ini
```

Hinweis: In der bereits vorhandenen „VADs_examples.ini“ Datei gibt es mehrere Konfigurationen, die alle vorimplementierten Beispiele ansprechen. Man sieht hier, dass

***.tier1Type* nur ganz unten in *[General]* definiert ist. Parameter in diesem Abschnitt werden, sofern sie noch nicht in der entsprechenden Konfiguration gesetzt sind, von hier übernommen.

Hinweis: In der vorimplementierten Datei wird ***.targetOverlayTerminalNum* als *${N=10}* festgelegt. Mit dieser Syntax wird für die Konfiguration eine Variable *N* erstellt, die dann bei anderen Parametern (wie hier ***.nodeNumber*) wiederverwendet werden kann. So können wir sicherstellen, dass diese zwei Parameter immer gleich sind. Bei Interesse bezüglich Parameter und ihrer Konfiguration kann man alles Wichtige im OMNeT++ Simulation Manual nachgelesen werden.

6.5 Ausführen der Simulation

Nun können wir eine Simulation mit der von uns festgelegten Konfiguration durchführen. Dazu machen wir einen Rechtsklick auf den Ordner „OverSim“ und wählen „Run As -> OMNeT++ Simulation“. Wir werden darum gebeten, eine .ini Datei anzugeben und wählen die von uns erstellte „VADs_example.ini“. Danach werden wir benachrichtigt, dass eine Run Configuration erstellt wurde. Diese können wir für zukünftige Simulationen einfach direkt starten. OMNeT++ fragt uns nun gegebenenfalls, welche Konfiguration wir ausführen wollen (falls in der .ini mehrere Konfigurationen angegeben sind). Es öffnen sich zwei Fenster. In einem wird während der Simulation unser Netzwerk graphisch dargestellt. Im Anderen sehen wir den Output der Simulation und eine Zeitleiste aller eingetretenen Ereignisse.

7 Hilfreiche Ressourcen

- Der OMNeT++ Install Guide für die Version 4.2.2 (in Sektion 8)
- OMNeT++ Website
- OMNeT++ Dokumentation und Tutorials (hilfreich zum tieferen Verständnis)
- OMNeT++ Simulation Manual (enthält hilfreiche Tipps z.B. zu Konfigurationen und zum Simulieren im Allgemeinen)
- OverSim Website
- OverSim Dokumentation (enthält Erläuterungen des Modells sowie Beispiele und Hinweise zur Vertiefung)
- Oracle VirtualBox Website
- tillk@mail.upb.de (Bei Fragen bezüglich OMNeT++ / OverSim und den Erweiterungen. Bitte mit „VADs OverSim“ im Betreff)

8 OMNeT++ Install Guide

OMNeT++

Installation Guide

Version 4.2.2

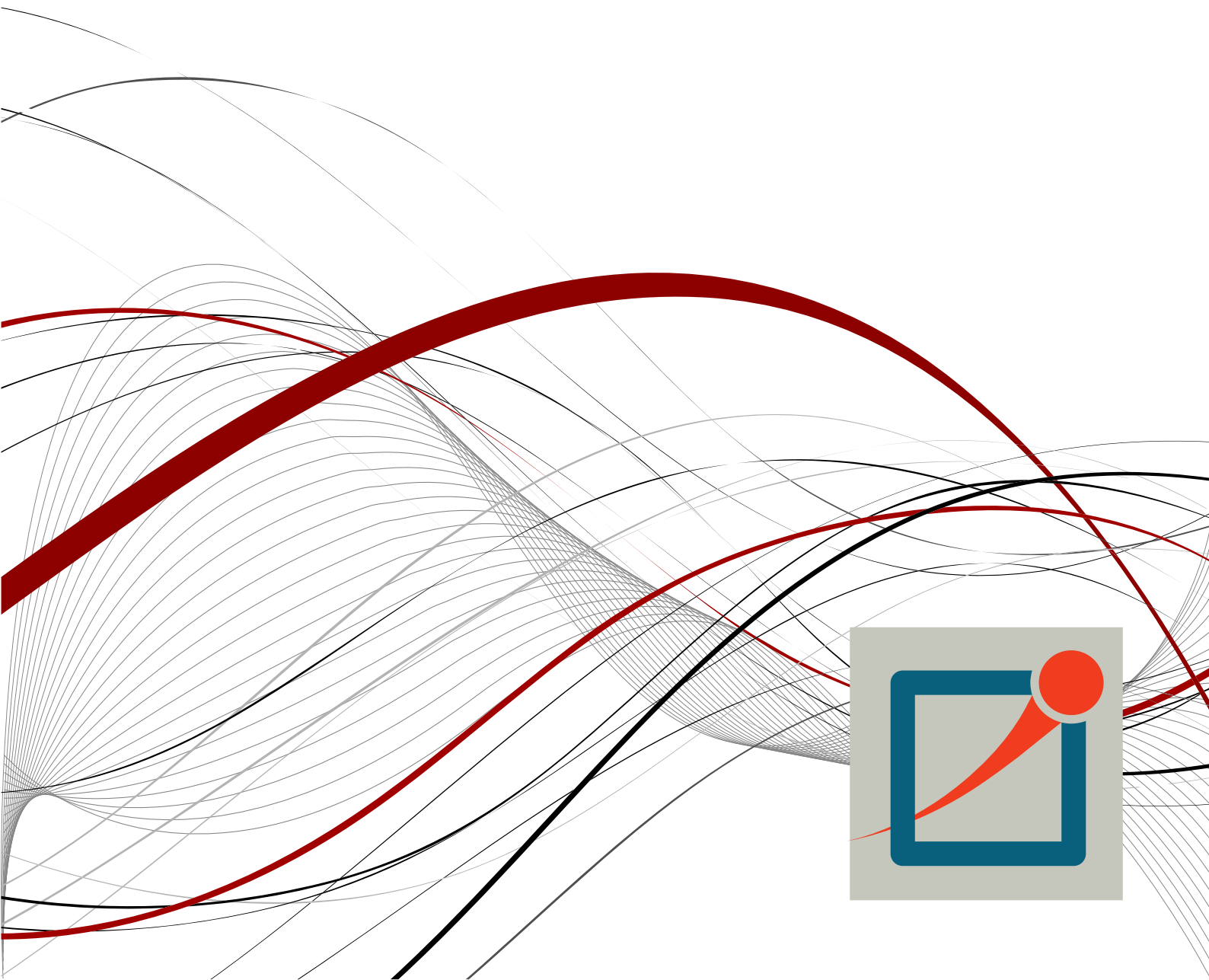


Table of Contents

1. General Information 1
2. Windows 2
3. Mac OS X 6
4. Linux 11
5. Ubuntu 17
6. Fedora 15 and 16 20
7. Red Hat 22
8. OpenSUSE 24
9. Generic Unix 26
10. Build Options 33

Chapter 1. General Information

1.1. Introduction

This document describes how to install OMNeT++ on various platforms. One chapter is dedicated to each operating system.

1.2. Supported Platforms

OMNeT++ has been tested and is supported on the following operating systems:

- Windows 7, Vista, XP
- Mac OS X 10.6 and 10.7
- Linux distributions covered in this Installation Guide

The Simulation IDE can be used on the following platforms:

- Linux x86
- Windows 7, Vista, XP
- Mac OS X 10.6 and 10.7

Simulations can be run practically on any unix-like environment with a decent and fairly up-to-date C++ compiler, for example gcc 4.x. Certain OMNeT++ features (Tkenv, parallel simulation, XML support, etc.) depend on the availability of external libraries (Tcl/Tk, MPI, LibXML or Expat, etc.)

IDE platforms are restricted because the IDE relies on a native shared library, which we compile for the above platforms and distribute in binary form for convenience.

Chapter 2. Windows

2.1. Supported Windows Versions

The supported Windows versions are the Intel 32-bit versions of Windows XP, and later Windows versions such as Vista and Windows 7.

2.2. Installing OMNeT++

Download the OMNeT++ source code from http://omnetpp.org. Make sure you select the Windows-specific archive, named `omnetpp-4.2.2-src-windows.zip`.

The package is nearly self-contained: in addition to OMNeT++ files it includes a C++ compiler, a command-line build environment, and all libraries and programs required by OMNeT++.

Copy the OMNeT++ archive to the directory where you want to install it. Choose a directory whose full path **does not contain any space**; for example, do not put OMNeT++ under *Program Files*.

Extract the zip file. To do so, right-click the zip file in Windows Explorer, and select *Extract All* from the menu. You can also use external programs like Winzip or 7zip. Rename the resulting directory to `omnetpp-4.2.2`.

When you look into the new `omnetpp-4.2.2` directory, should see directories named `doc`, `images`, `include`, `msys`, etc., and files named `mingwenv.cmd`, `configure`, `Makefile`, and others.

2.3. Configuring and Building OMNeT++

Start `mingwenv.cmd` in the `omnetpp-4.2.2` directory by double-clicking it in Windows Explorer. It will bring up a console with the MSYS *bash* shell, where the path is already set to include the `omnetpp-4.2.2/bin` directory.

If you want to start simulations from outside the shell as well (for example from Explorer), you need to add OMNeT++'s `bin` directory to the path; instructions are provided later.

First, check the contents of the `configure.user` file to make sure it contains the settings you need. In most cases you don't need to change anything.

```
notepad configure.user
```

Then enter the following commands:

```
$ ./configure
$ make
```

The build process will create both debug and release binaries.

2.4. Verifying the Installation

You should now test all samples and check they run correctly. As an example, the *dyna* example is started by entering the following commands:

```
$ cd samples/dyna
$ ./dyna
```

By default, the samples will run using the graphical Tkenv environment. You should see GUI windows and dialogs.

2.5. Starting the IDE

OMNeT++ comes with an Eclipse-based Simulation IDE. You should be able to start the IDE by typing:

```
$ omnetpp
```

We recommend that you create a shortcut for starting the IDE. To do so, locate the `omnetpp.exe` program in the `omnetpp-4.2.2/ide` directory in Windows Explorer, right-click it, and choose *Send To > Desktop (create shortcut)* from the menu. On Windows 7, you can right-click the taskbar icon while the IDE is running, and select *Pin this program to taskbar* from the context menu.

When you try to build a project in the IDE, you may get the following warning message:

> Toolchain "..." is not supported on this platform or installation. Please go to the Project menu, and activate a different build configuration. (You may need to switch to the C/C++ perspective first, so that the required menu items appear in the Project menu.)

If you encounter this message, choose *Project > Properties > C/C++ Build > Tool Chain Editor > Current toolchain > GCC for OMNeT++*.

2.6. Environment Variables

If you want to start OMNeT++ simulations outside the shell as well (for example from Exlorer), you need to add OMNeT++'s `bin` directory to the path.

First, open the *Environment Variables* dialog.

- On Windows XP and Vista: Right-click *My Computer*, and choose *Properties > Advanced > Environment variables*.
- On Windows 7: Click the Start button, then start typing `environment variables` into the search box. Choose *Edit environment variables for your account* when it appears in the list. The dialog comes up.

In the dialog, select `path` or `PATH` in the list, click *Edit*. Append "`;<omnetpp-dir>\bin`" to the value (without quotes), where `<omnetpp-dir>` is the name of the OMNeT++ root directory (for example `C:\omnetpp-4.2.2`). Hit Enter to accept.

You need to close and re-open any command windows for the changes to take effect in them.

2.7. Reconfiguring the Libraries

If you need to recompile the OMNeT++ components with different flags (e.g. different optimization), then change the top-level OMNeT++ directory, edit `configure.user` accordingly, then type:

```
$ ./configure
$ make clean
$ make
```

If you want to recompile just a single library, then change to the directory of the library (e.g. `cd src/sim`) and type:

```
$ make clean
$ make
```

By default, libraries are compiled in both debug and release mode. If you want to make release or debug builds only, use:

```
$ make MODE=release
```

or

```
$ make MODE=debug
```

By default, shared libraries will be created. If you want to build static libraries, set `SHARED_LIBS=no` in `configure.user` and re-configure your project.

The built libraries and programs are immediately copied to the `lib/` and `bin/` subdirs.

2.8. Portability Issues

OMNeT++ has been tested with the MinGW gcc compiler. The current distribution contains gcc version 4.6.

Microsoft Visual C++ is not supported in the Academic Edition.

2.9. Additional Packages

Note that Doxygen and GraphViz are already included in the OMNeT++ package, and do not need to be downloaded.

2.9.1. MPI

MPI is only needed if you would like to run parallel simulations.

There are several MPI implementations for Windows, and OMNeT++ does not mandate any specific one. We recommend DeinoMPI, which can be downloaded from http://mpi.deino.net.

After installing DeinoMPI, adjust the `MPI_DIR` setting in OMNeT++'s `configure.user`, and reconfigure and recompile OMNeT++:

```
$ ./configure
$ make cleanall
$ make
```

In general, if you would like to run parallel simulations, we recommend that you use Linux, OS X, or another unix-like platform.

2.9.2. PCAP

The optional WinPcap library allows simulation models to capture and transmit network packets bypassing the operating system's protocol stack. It is not used directly by OMNeT++, but OMNeT++ detects the necessary compiler and linker options for models in case they need it.

2.9.3. Akaroa

Akaroa 2.7.9, which is the latest version at the time of writing, does not support Windows. You may try to port it using the porting guide from the Akaroa distribution.

Chapter 3. Mac OS X

3.1. Supported Releases

This chapter provides additional information for installing OMNeT++ on Mac OS X.

The following releases are covered:

- Mac OS X 10.6 (*Snow Leopard*)
- Mac OS X 10.7 (*Lion*)

They were tested on the following architectures:

- Intel 32/64-bit

3.2. Installing the Prerequisite Packages

Mac OS X 10.6 (Snow Leopard)

- Install Xcode 3.2.x on your machine. It can be downloaded from http://developer.apple.com/xcode/.

Mac OS X 10.7 (Lion)

- Install Xcode 4.3 or later on your machine. It can be downloaded from the App Store.
- You MUST activalte command line support by starting XCode and installing `Command Line Tools` from `Xcode | Preferences | Downloads | Components`.
- Install the Java Runtime from http://support.apple.com/kb/DL1421 , because OS X Lion does not provide it by default.

Installing additional packages will enable more functionality in OMNeT++; see the *Additional packages* section at the end of this chapter.

3.3. Downloading and Unpacking OMNeT++

Download OMNeT++ from http://omnetpp.org. Make sure you select to download the generic archive, `omnetpp-4.2.2-src.tgz`.

Copy the archive to the directory where you want to install it. This is usually your home directory, `/Users/<you>`. Open a terminal, and extract the archive using the following command:

```
$ tar zxvf omnetpp-4.2.2-src.tgz
```

A subdirectory called `omnetpp-4.2.2` will be created, containing the simulator files.

Alternatively, you can also unpack the archive using Finder.

The Terminal can be found in the Applications / Utilities folder.

3.4. Environment Variables

OMNeT++ needs its `bin/` directory to be in the path. To add `bin/` to `PATH` temporarily (in the current shell only), change into the OMNeT++ directory and source the `setenv` script:

```
$ cd omnetpp-4.2.2
$ . setenv
```

To set the environment variables permanently, edit `.bashrc` in your home directory. Use your favourite text editor to edit `.bashrc`, for example TextEdit:

```
$ touch ~/.bashrc
$ open -e ~/.bashrc
```

`touch` is needed because `open -e` only opens existing files. Alternatively, you can use the terminal-based *pico* editor (`pico ~/.bashrc`)

Add the following line at the end of the file, then save it:

```
export PATH=$PATH:$HOME/omnetpp-4.2.2/bin
```

You need to close and re-open the terminal for the changes to take effect.

Alternatively, you can put the above line into `~/.bash_profile`, but then you need to log out and log in again for the changes to take effect.

If you use a shell other than the default one, *bash*, consult the man page of that shell to find out which startup file to edit, and how to set and export variables.

3.5. Configuring and Building OMNeT++

Check `configure.user` to make sure it contains the settings you need. In most cases you don't need to change anything in it.

In the top-level OMNeT++ directory, type:

```
$ ./configure
```

The `configure` script detects installed software and configuration of your system. It writes the results into the `Makefile.inc` file, which will be read by the makefiles during the build process.

Normally, the `configure` script needs to be running under the graphical environment in order to test for `wish`, the Tcl/Tk shell. If you are logged in via an ssh session or you want to compile OMNeT++ without Tcl/Tk, use the command

```
$ NO_TCL=1 ./configure
```

instead of plain `./configure`.

If there is an error during `configure`, the output may give hints about what went wrong. Scroll up to see the messages. (You may need to increase the scrollback buffer size of the terminal and re-run `./configure`.) The script also writes a very detailed log of its operation into `config.log` to help track down errors. Since `config.log` is very long, it is recommended that you open it in an editor and search for phrases like *error* or the name of the package associated with the problem.

OMNeT++ builds without the BLT Tcl/Tk extension, because BLT is not available for the native (Aqua) version of Tcl/Tk.

As an alternative to the Aqua version of Tcl/Tk, you may also use the X11 version from the MacPorts repository (http://macports.org). The MacPorts version does not have the look and feel of OS X applications, but it may provide better performance with animations. To select the MacPorts version of Tk, you will need to adjust the Tcl/Tk settings in OMNeT++'s `configure.user` file and re-run the `./configure`. Please note that we do not provide support for using the MacPorts version of Tcl/Tk.

When `./configure` has finished, you can compile OMNeT++. Type in the terminal:

```
$ make
```

To take advantage of multiple processor cores, add the `-j2` option to the `make` command line.

The build process will not write anything outside its directory, so no special privileges are needed.

The make command will seemingly compile everything twice. This is because both debug and optimized versions of the libraries are built. If you only want to build one set of the libraries, specify `MODE=debug` or `MODE=release`:

```
$ make MODE=release
```

3.6. Verifying the Installation

You can now verify that the sample simulations run correctly. For example, the dyna simulation is started by entering the following commands:

```
$ cd samples/dyna
$ ./dyna
```

By default, the samples will run using the Tcl/Tk environment. You should see nice gui windows and dialogs.

3.7. Starting the IDE

OMNeT++ comes with an Eclipse-based Simulation IDE. Mac OS X 10.6 already has Java installed, so you do not need to install it. On Mac OS X 10.7 (Lion), the Java Runtime must be installed (see prerequisites) before you can use the IDE. Start the IDE by typing:

```
$ omnetpp
```

If you would like to be able to launch the IDE via Applications, the Dock or a desktop shortcut, do the following: open the `omnetpp-4.2.2` folder in Finder, go into the `ide` subfolder, create an alias for the omnetpp program there (right-click, *Make Alias*), and drag the new alias into the Applications folder, onto the Dock, or onto the desktop.

Alternatively, run one or both of the commands below:

```
$ make install-menu-item
$ make install-desktop-icon
```

which will do roughly the same.

3.8. Using the IDE

When you try to build a project in the IDE, you may get the following warning message:

> Toolchain "..." is not supported on this platform or installation. Please go to the Project menu, and activate a different build configuration. (You may need to switch to the C/C++ perspective first, so that the required menu items appear in the Project menu.)

If you encounter this message, choose *Project* > *Properties* > *C/C++ Build* > *Tool Chain Editor* > *Current toolchain* > *GCC for OMNeT++*.

The IDE is documented in detail in the *User Guide*.

3.9. Reconfiguring the Libraries

If you need to recompile the OMNeT++ components with different flags (e.g. different optimization), then change the top-level OMNeT++ directory, edit `configure.user` accordingly, then type:

```
$ ./configure
$ make clean
$ make
```

To take advantage of multiple processor cores, add the `-j2` option to the `make` command line.

If you want to recompile just a single library, then change to the directory of the library (e.g. `cd src/sim`) and type:

```
$ make clean
$ make
```

By default, libraries are compiled in both debug and release mode. If you want to make release or debug builds only, use:

```
$ make MODE=release
```

or

```
$ make MODE=debug
```

By default, shared libraries will be created. If you want to build static libraries, set `SHARED_LIBS=no` in `configure.user` and re-configure your project.

The built libraries and programs are immediately copied to the `lib/` and `bin/` subdirectories.

The Tcl/Tk environment uses the native Aqua version of Tcl/Tk, so you will see native widgets. However, due to problems in the Tk/Aqua port, you may experience minor UI quirks. We are aware of these problems, and are working on the solution.

3.10. Additional Packages

3.10.1. OpenMPI

- OpenMPI is already bundled with OS X 10.6, so you do not need to manually install it.

- OS X 10.7 does not come with OpenMPI, so you must install it manually. Download it from http://open-mpi.org and follow the installation instructions. Alternatively, you can install it from the MacPorts repo by typing `sudo port install openmpi`. In this case, you have to manually set the MPI_CFLAGS and MPI_LIBS variables in `configure.user` and re-run `./configure`. Please note that we do not provide support for OpenMPI installed from the MacPorts repository.

MacPorts is a repository of several open source packages for Mac OS X. Using MacPorts packages may save you some manual work. You can install MacPorts from http://www.macports.org.

3.10.2. GraphViz

GraphViz is needed if you want to have diagrams in HTML documentation that you generate from NED files in the IDE (*Generate NED Documentation...* item in the project context menu).

Download and install the GraphViz OS X binaries from http://www.pixelglow.com/graphviz/download/. Download the latest, version 2.x package; at the time of writing, the link is at the top of the page.

Alternatively, you can install it from the MacPorts project by typing `sudo port install graphviz`.

After installation, make sure that the `dot` program is available from the command line. Open a terminal, and type

```
$ dot -V
```

Note the capital *V*. The command should normally work out of the box. If you get the "*command not found*" error, you need to put `dot` into the path. Find the `dot` program in the GraphViz installation directory, and soft link it into `/usr/local/bin` (`sudo ln -s <path>/dot /usr/local/bin`).

3.10.3. Doxygen

Doxygen is needed if you want to generate documentation for C++ code, as part of the HTML documentation that you generate from NED files in the IDE (*Generate NED Documentation...* item in the project context menu).

Download the Doxygen OS X binaries from the Doxygen web site's download page, http://www.stack.nl/~dimitri/doxygen/download.html, and install it.

Alternatively, you can install it from the MacPorts project by typing `sudo port install doxygen`.

After installation, ensure that the `doxygen` program is available from the command line. Open a terminal, and type

```
$ doxygen
```

If you get the "*command not found*" error, you need to put `doxygen` into the path. Enter into a terminal:

```
$ cd /Applications/Doxygen.app/Contents/Resources/
$ sudo ln -s doxygen doxytags /usr/local/bin
```

3.10.4. Akaroa

Akaroa 2.7.9, which is the latest version at the time of writing, does not support Mac OS X. You may try to port it using the porting guide from the Akaroa distribution.

Chapter 4. Linux

4.1. Supported Linux Distributions

This chapter provides instructions for installing OMNeT++ on selected Linux distributions:

- Ubuntu 10.04 LTS, 11.04
- Fedora Core 15, 16
- Red Hat Enterprise Linux Desktop Workstation 5.5
- OpenSUSE 11.4

This chapter describes the overall process. Distro-specific information, such as how to install the prerequisite packages, are covered by distro-specific chapters.

If your Linux distribution is not listed above, you still may be able to use some distro-specific instructions in this Guide.

Ubuntu derivatives (Ubuntu instructions may apply):

- Kubuntu, Xubuntu, Edubuntu, ...
- Linux Mint

Some Debian-based distros (Ubuntu instructions may apply, as Ubuntu itself is based on Debian):

- Knoppix and derivatives
- Mepis

Some Fedora-based distros (Fedora instructions may apply):

- Simplis
- Eeedora

4.2. Installing the Prerequisite Packages

OMNeT++ requires several packages to be installed on the computer. These packages include the C++ compiler (gcc), the Java runtime, and several other libraries and programs. These packages can be installed from the software repositories of your Linux distribution.

See the chapter specific to your Linux distribution for instructions on installing the packages needed by OMNeT++.

You may need superuser permissions to install packages.

Not all packages are available from software repositories; some (optional) ones need to be downloaded separately from their web sites, and installed manually. See the section *Additional Packages* later in this chapter.

4.3. Downloading and Unpacking

Download OMNeT++ from http://omnetpp.org. Make sure you select to download the generic archive, `omnetpp-4.2.2-src.tgz`.

Copy the archive to the directory where you want to install it. This is usually your home directory, `/home/<you>`. Open a terminal, and extract the archive using the following command:

```
$ tar xvfz omnetpp-4.2.2-src.tgz
```

This will create an `omnetpp-4.2.2` subdirectory with the OMNeT++ files in it.

On how to open a terminal on your Linux installation, see the chapter specific to your Linux distribution.

4.4. Environment Variables

OMNeT++ needs its `bin/` directory to be in the path. To add `bin/` to `PATH` temporarily (in the current shell only), change into the OMNeT++ directory and source the `setenv` script:

```
$ cd omnetpp-4.2.2
$ . setenv
```

The script also adds the `lib/` subdirectory to `LD_LIBRARY_PATH`, which may be necessary on systems that don't support the rpath mechanism.

To set the environment variables permanently, edit `.bashrc` in your home directory. Use your favourite text editor to edit `.bashrc`, for example gedit:

```
$ gedit ~/.bashrc
```

Add the following line at the end of the file, then save it:

```
export PATH=$PATH:$HOME/omnetpp-4.2.2/bin
```

You need to close and re-open the terminal for the changes to take effect.

Alternatively, you can put the above line into `~/.bash_profile`, but then you need to log out and log in again for the changes to take effect.

If you use a shell other than *bash*, consult the man page of that shell to find out which startup file to edit, and how to set and export variables.

Note that all Linux distributions covered in this Installation Guide use *bash* unless the user has explicitly selected another shell.

4.5. Configuring and Building OMNeT++

In the top-level OMNeT++ directory, type:

```
$ ./configure
```

The `configure` script detects installed software and configuration of your system. It writes the results into the `Makefile.inc` file, which will be read by the makefiles during the build process.

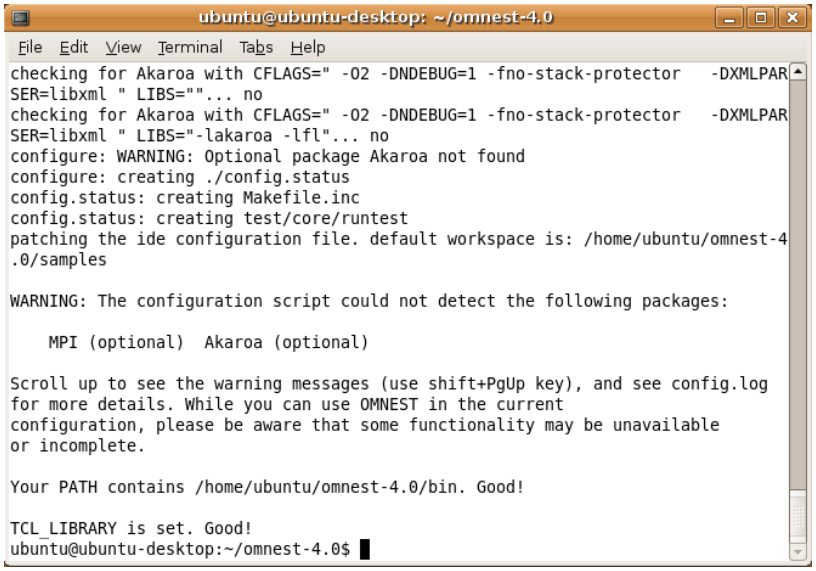


Figure 4.1. Configuring OMNeT++

Normally, the `configure` script needs to be running under the graphical environment (X11) in order to test for `wish`, the Tcl/Tk shell. If you are logged in via an *ssh* session, or there is some other reason why X is not running, the easiest way to work around the problem is to tell OMNeT++ to build without Tcl/Tk. To do that, use the command

```
$ NO_TCL=1 ./configure
```

instead of plain `./configure`.

If there is an error during `configure`, the output may give hints about what went wrong. Scroll up to see the messages. (Use Shift+PgUp; you may need to increase the scrollback buffer size of the terminal and re-run `./configure`.) The script also writes a very detailed log of its operation into `config.log` to help track down errors. Since `config.log` is very long, it is recommended that you open it in an editor and search for phrases like *error* or the name of the package associated with the problem.

When `./configure` has finished, you can compile OMNeT++. Type in the terminal:

```
$ make
```

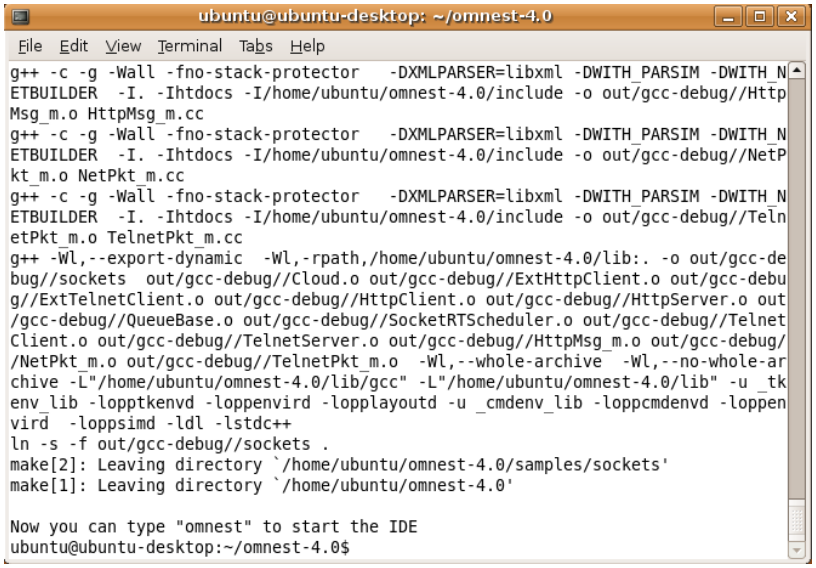


Figure 4.2. Building OMNeT++

To take advantage of multiple processor cores, add the `-j2` option to the `make` command line.

The build process will not write anything outside its directory, so no special privileges are needed.

The make command will seemingly compile everything twice. This is because both debug and optimized versions of the libraries are built. If you only want to build one set of the libraries, specify `MODE=debug` or `MODE=release`:

```
$ make MODE=release
```

4.6. Verifying the Installation

You can now verify that the sample simulations run correctly. For example, the dyna simulation is started by entering the following commands:

```
$ cd samples/dyna
$ ./dyna
```

By default, the samples will run using the Tcl/Tk environment. You should see nice gui windows and dialogs.

4.7. Starting the IDE

You can launch the OMNeT++ Simulation IDE by typing the following command in the terminal:

```
$ omnetpp
```

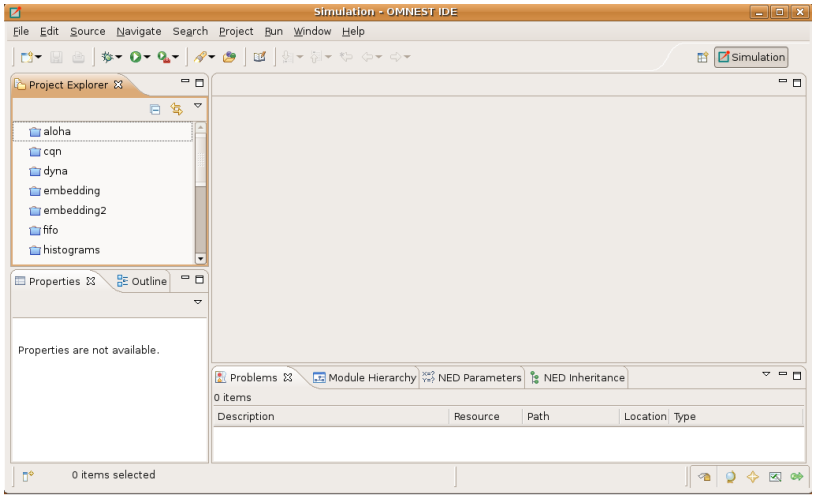


Figure 4.3. The Simulation IDE

If you would like to be able to access the IDE from the application launcher or via a desktop shortcut, run one or both of the commands below:

```
$ make install-menu-item
$ make install-desktop-icon
```

Or add a shortcut that points to the `omnetpp` program in the `ide` subdirectory by other means, for example using the Linux desktop's context menu.

4.8. Using the IDE

When you try to build a project in the IDE, you may get the following warning message:

> Toolchain "..." is not supported on this platform or installation. Please go to the Project menu, and activate a different build configuration. (You may need to switch to the C/C++ perspective first, so that the required menu items appear in the Project menu.)

If you encounter this message, choose *Project > Properties > C/C++ Build > Tool Chain Editor > Current toolchain > GCC for OMNeT++*.

The IDE is documented in detail in the *User Guide*.

4.9. Reconfiguring the Libraries

If you need to recompile the OMNeT++ components with different flags (e.g. different optimization), then change the top-level OMNeT++ directory, edit `configure.user` accordingly, then type:

```
$ ./configure
$ make cleanall
$ make
```

If you want to recompile just a single library, then change to the directory of the library (e.g. `cd src/sim`) and type:

```
$ make clean
$ make
```

By default, libraries are compiled in both debug and release mode. If you want to make release or debug builds only, use:

```
$ make MODE=release
```

or

```
$ make MODE=debug
```

By default, shared libraries will be created. If you want to build static libraries, set `SHARED_LIBS=no` in `configure.user` and re-configure your project.

For detailed description of all options please read the *Build Options* chapter.

4.10. Additional Packages

Note that at this point, MPI, Doxygen and GraphViz have been installed as part of the prerequisites.

4.10.1. Akaroa

Linux distributions do not contain the Akaroa package. It must be downloaded, compiled and installed manually before installing OMNeT++.

As of version 2.7.9, Akaroa only supports Linux and Solaris.

Download Akaroa 2.7.9 from: http://www.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_group/akaroa/download.chtml

Extract it into a temporary directory:

```
$ tar xfz akaroa-2.7.9.tar.gz
```

Configure, build and install the Akaroa library. By default, it will be installed into the `/usr/local/akaroa` directory.

```
$ ./configure
$ make
$ sudo make install
```

Go to the OMNeT++ directory, and (re-)run the `configure` script. Akaroa will be automatically detected if you installed it to the default location.

4.10.2. PCAP

The optional Pcap library allows simulation models to capture and transmit network packets bypassing the operating system's protocol stack. It is not used directly by OMNeT++, but OMNeT++ detects the necessary compiler and linker options for models in case they need it.

Chapter 5. Ubuntu

5.1. Supported Releases

This chapter provides additional information for installing OMNeT++ on Ubuntu Linux installations. The overall installation procedure is described in the *Linux* chapter.

The following Ubuntu releases are covered:

- Ubuntu 10.04 LTS
- Ubuntu 11.04

They were tested on the following architectures:

- Intel 32-bit and 64-bit

The instructions below assume that you use the Gnome desktop and the bash shell, which are the defaults. If you use another desktop environment or shell, you may need to adjust the instructions accordingly.

5.2. Opening a Terminal

Choose *Applications* > *Accessories* > *Terminal* from the menu.

This only applies if you use the Gnome Desktop (default).

5.3. Installing the Prerequisite Packages

You can perform the installation using the graphical user interface or from the terminal, whichever you prefer.

5.3.1. Command-Line Installation

Before starting the installation, refresh the database of available packages. Type in the terminal:

```
$ sudo apt-get update
```

To install the required packages, type in the terminal:

```
$ sudo apt-get install build-essential gcc g++ bison flex perl \
      tcl-dev tk-dev blt libxml2-dev zlib1g-dev openjdk-6-jre \
      doxygen graphviz openmpi-bin libopenmpi-dev libpcap-dev
```

At the confirmation questions (*Do you want to continue? [Y/N]*), answer *Y*.

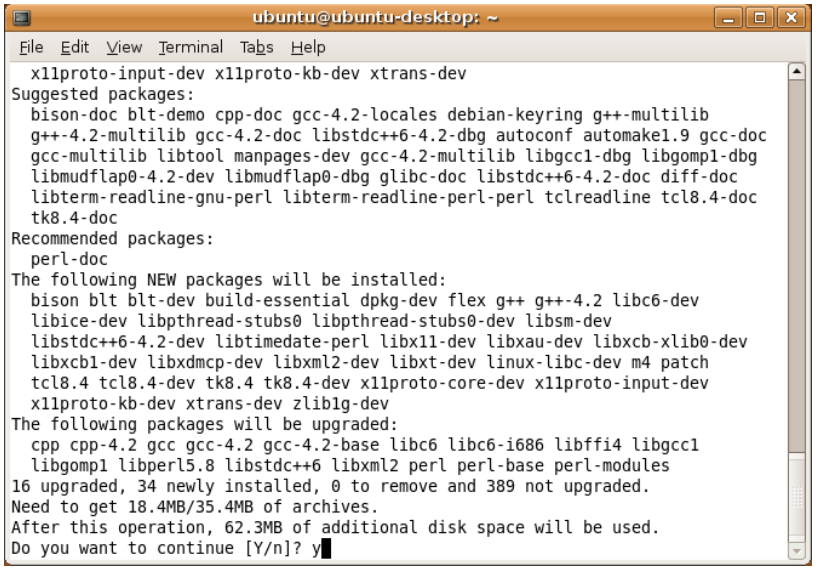


Figure 5.1. Command-Line Package Installation

5.3.2. Graphical Installation

Ubuntu's graphical installer, *Synaptic*, can be started with the *System* > *Administration* > *Synaptic package manager* menu item.

Since software installation requires root permissions, Synaptic will ask you to type your password.

Search for the following packages in the list, click the squares before the names, then choose *Mark for installation* or *Mark for upgrade*.

If the *Mark additional required changes?* dialog comes up, choose the *Mark* button.

The packages:

- build-essential, gcc, g++, bison, flex, perl, tcl-dev, tk-dev, blt, libxml2-dev, zlib1g-dev, openjdk-6-jre, doxygen, graphviz, openmpi-bin, libopenmpi-dev, libpcap-dev

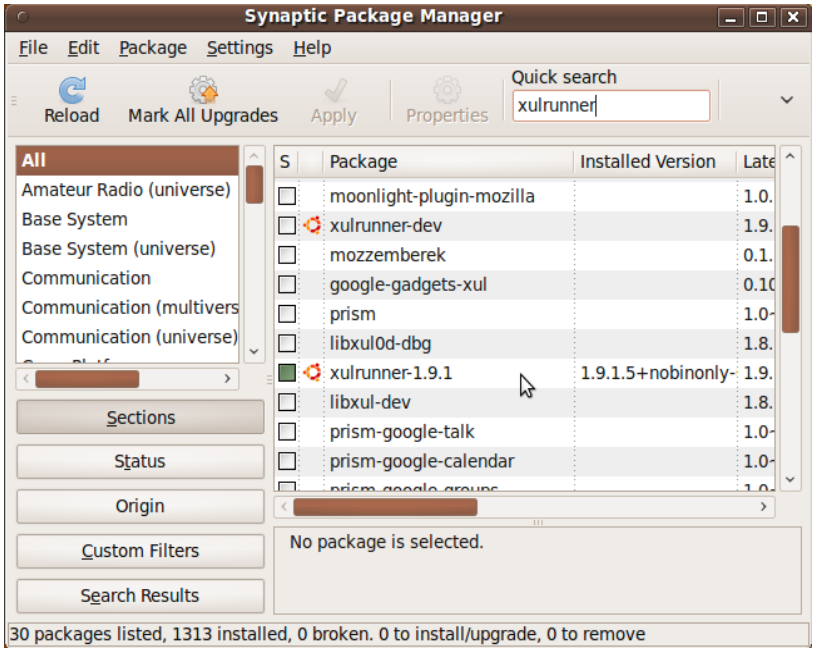


Figure 5.2. Synaptic Package Manager

Click *Apply*, then in the *Apply the following changes?* window, click *Apply* again. In the *Changes applied* window, click *Close*.

Chapter 6. Fedora 15 and 16

6.1. Supported Releases

This chapter provides additional information for installing OMNeT++ on Fedora installations. The overall installation procedure is described in the *Linux* chapter.

The following Fedora releases are covered:

- Fedora 15 and 16

They were tested on the following architectures:

- Intel 32-bit and 64-bit

6.2. Opening a Terminal

Choose *Applications* > *System Tools* > *Terminal* from the menu.

6.3. Installing the Prerequisite Packages

You can perform the installation using the graphical user interface or from the terminal, whichever you prefer.

6.3.1. Command-Line Installation

To install the required packages, type in the terminal:

```
$ su -c 'yum install make gcc gcc-c++ bison flex perl \
    tcl-devel tk-devel blt libxml2-devel zlib-devel \
    java-1.6.0-openjdk doxygen graphviz openmpi-devel libpcap-devel'
```

then follow the instruction on the console.

Note that *openmpi* will not be available by default, it needs to be activated in every session with the

```
$ module load openmpi-<arch>
```

command, where `<arch>` is your architecture (usually `i386` or `x86_64`). When in doubt, use `module avail` to display the list of available modules. If you need MPI in every session, you may add the `module load` command to your startup script (`.bashrc`).

6.3.2. Graphical Installation

The graphical installer can be launched by choosing *System* > *Administration* > *Add/Remove Software* from the menu.

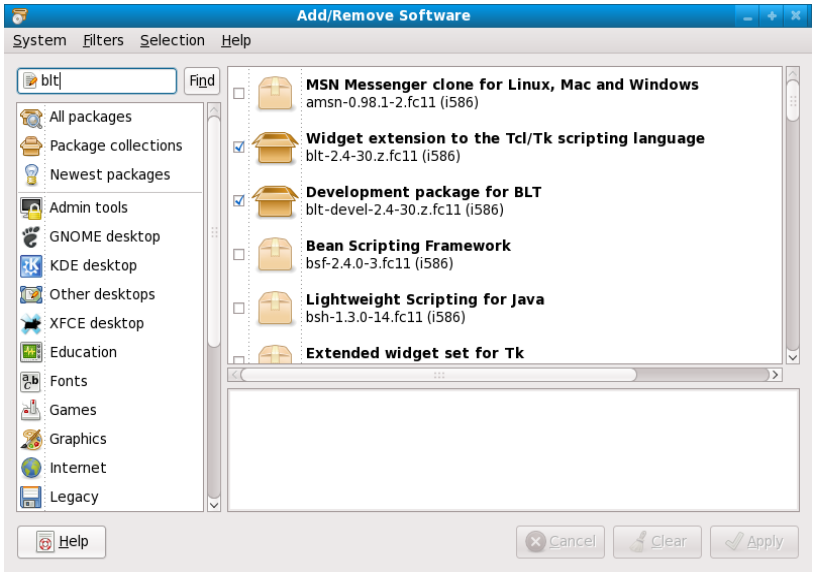


Figure 6.1. Add/Remove Software

Search for the following packages in the list. Select the checkboxes in front of the names, and pick the latest version of each package.

The packages:

- bison, gcc, gcc-c++, flex, perl, tcl-devel, tk-devel, blt, libxml2-devel, zlib-devel, xulrunner, make, java-1.6.0-openjdk, doxygen, graphviz, openmpi-devel, libpcap-devel

Click *Apply*, then follow the instructions.

Chapter 7. Red Hat

7.1. Supported Releases

This chapter provides additional information for installing OMNeT++ on Red Hat Enterprise Linux installations. The overall installation procedure is described in the *Linux* chapter.

The following Red Hat release is covered:

- Red Hat Enterprise Linux Desktop Workstation 5.5

It was tested on the following architectures:

- Intel 32-bit

7.2. Opening a Terminal

Choose *Applications* > *Accessories* > *Terminal* from the menu.

7.3. Installing the Prerequisite Packages

You can perform the installation using the graphical user interface or from the terminal, whichever you prefer.

You will need Red Hat Enterprise Linux Desktop Workstation for OMNeT++. The *Desktop Client* version does not contain development tools.

7.3.1. Command-Line Installation

To install the required packages, type in the terminal:

```
$ su -c 'yum install make gcc gcc-c++ bison flex perl \
    tcl-devel tk-devel libxml2-devel zlib-devel \
    java-1.6.0-openjdk doxygen openmpi-devel libpcap-devel'
```

then follow the instruction on the console.

Note that *openmpi* will not be available by default, it needs to be activated in every session with the

```
$ module load openmpi_<arch>
```

command, where `<arch>` is your architecture (usually `i386` or `x86_64`). When in doubt, use `module avail` to display the list of available modules. If you need MPI in every session, you may add the `module load` command to your startup script (`.bashrc`).`

Red Hat provides no BLT package, so it is missing from the above list. BLT is optional in OMNeT++ and its absence will not result in loss of functionality, but you can download a BLT source package from sourceforge.net, and compile and install it yourself if you wish. See the *Generic Unix* chapter of this Installation Guide for more

information on BLT. Alternatively you may search for binary RPM packages using `http://rpm.pbone.net`

GraphViz packages for RHEL can be installed from graphviz.org, see http://www.graphviz.org/Download_linux_rhel.php. GraphViz is an optional package for OMNeT++; you only need it if you plan to generate HTML documentation from NED files, and would like the resulting documentation to contain diagrams.

7.3.2. Graphical Installation

The graphical installer can be launched by choosing *Applications* > *Add/Remove Software* from the menu.

Search for the following packages in the list. Select the checkboxes in front of the names, and pick the latest version of each package.

The packages:

- gcc, gcc-c++, bison, flex, perl, tcl-devel, tk-devel, libxml2-devel, zlib-devel, make, java-1.6.0-openjdk, doxygen, openmpi-devel, libpcap-devel

Click *Apply*, then follow the instructions.

The above list does not contain BLT and GraphViz; see the previous section for more info about these packages.

7.4. SELinux

You may need to turn off SELinux when running certain simulations. To do so, click on *System* > *Administration* > *Security Level* > *Firewall*, go to the *SELinux* tab, and choose *Disabled*.

You can verify the SELinux status by typing the `sestatus` command in a terminal.

From OMNeT++ 4.1 on, makefiles that build shared libraries include the `chcon -t textrel_shlib_t lib<name>.so` command that properly sets the security context for the library. This should prevent the SELinux-related "*cannot restore segment prot after reloc: Permission denied*" error from occurring, unless you have a shared library which was built using an obsolete or hand-crafted makefile that does not contain the `chcon` command.

Chapter 8. OpenSUSE

8.1. Supported Releases

This chapter provides additional information for installing OMNeT++ on openSUSE installations. The overall installation procedure is described in the *Linux* chapter.

The following openSUSE releases are covered:

- openSUSE 11.4

They were tested on the following architectures:

- Intel 32-bit and 64-bit

8.2. Opening a Terminal

Choose *Applications* > *System* > *Terminal* > *Terminal* from the menu.

8.3. Installing the Prerequisite Packages

You can perform the installation using the graphical user interface or from the terminal, whichever you prefer.

8.3.1. Command-Line Installation

To install the required packages, type in the terminal:

```
$ sudo zypper install make gcc gcc-c++ bison flex perl \
    tcl-devel tk-devel libxml2-devel zlib-devel \
    java-1_6_0-openjdk doxygen graphviz openmpi-devel libpcap-devel \
    mozilla-xulrunner192
```

then follow the instruction on the console.

Note that *openmpi* will not be available by default, first you need to log out and log in again, or source your `.profile` script:

```
$ . ~/.profile
```

OpenSUSE provides no BLT package, so it is missing from the above list. BLT is optional in OMNeT++ and its absence will not result in loss of functionality, but you can download a BLT source package from sourceforge.net, and compile and install it yourself if you wish. See the *Generic Unix* chapter of this Installation Guide for more information on BLT.

8.3.2. Graphical Installation

The graphical installer can be launched by choosing *Computer* > *Yast* > *System* > *Software* > *Software Management* from the menu.

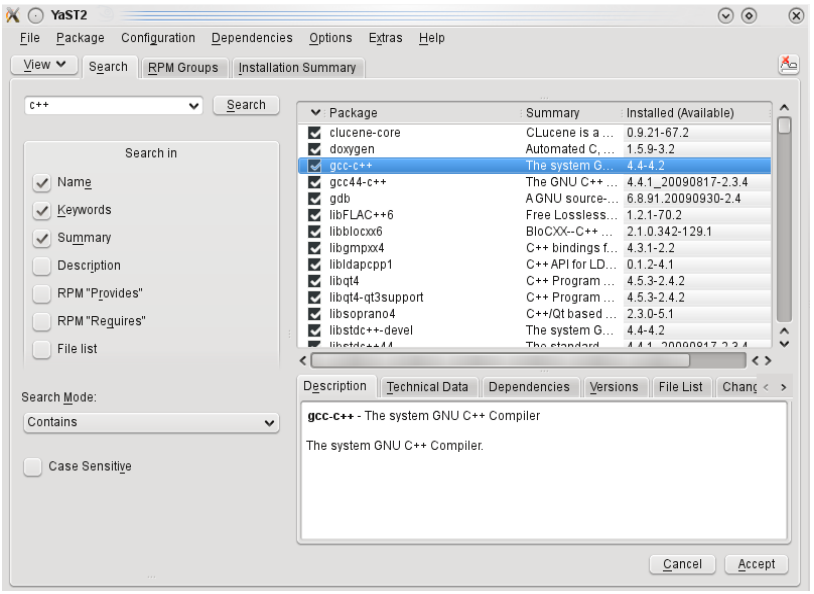


Figure 8.1. Yast Software Management

Search for the following packages in the list. Select the checkboxes in front of the names, and pick the latest version of each package.

The packages:

- make, gcc, gcc-c++, bison, flex, perl, tcl-devel, tk-devel, libxml2-devel, zlib-devel, java-1_6_0-openjdk, doxygen, graphviz, openmpi-devel, libpcap-devel

Click *Accept*, then follow the instructions.

Chapter 9. Generic Unix

9.1. Introduction

This chapter provides additional information for installing OMNeT++ on Unix-like operating systems not specifically covered by this Installation Guide. The list includes FreeBSD, Solaris, and Linux distributions not covered in other chapters.

In addition to Windows and Mac OS X, the Simulation IDE will only work on Linux x86 32/64-bit platforms. Other operating systems (FreeBSD, Solaris, etc.) and architectures may still be used as simulation platforms, without the IDE.

9.2. Dependencies

The following packages are required for OMNeT++ to work:

build-essential, GNU make, gcc, g++, bison (2.x+), flex, perl	These packages are needed for compiling OMNeT++ and simulation models, and also for certain OMNeT++ tools to work. Some C++ compilers other than g++, for example the Intel compiler, will also be accepted.

The following packages are strongly recommended, because their absence results in severe feature loss:

Tcl/Tk 8.4 or later	Required by the Tkenv simulation runtime environment. You need the *devel* packages that include the C header files as well. It is also possible to compile OMNeT++ without Tcl/Tk (and Tkenv), by turning on the `NO_TCL` environment variable.
BLT (2.4z)	This is a Tk extension recommended for Tkenv. The normal (non-devel) package suffices. This package is optional in the sense that if it is missing, Tkenv will be still fully functional, albeit it will have a less sophisticated user interface.
LibXML2 or Expat	Either one of these XML parsers are needed for OMNeT++ to be able to read XML files. The *devel* packages (that include the header files) are needed. LibXML2 is the preferred one.
SUN JRE or OpenJDK, version 5.0 or later	The Java runtime is required to run the Eclipse-based Simulation IDE. Other implementations, for example Kaffe, have been found to have problems running the IDE. You do not need this package if you do not plan to use the Simulation IDE.
xulrunner	This package is part of Firefox, and is needed by the IDE to display documentation, styled tooltips and other items.

The following packages are required if you want to take advantage of some advanced OMNeT++ features:

GraphViz, Doxygen	These packages are used by the NED documentation generation feature of the IDE. When they are missing, documentation will have less content.
MPI	openmpi or some other MPI implementation is required to support parallel simulation execution.
Akaroa	Implements Multiple Replications In Parallel (MRIP). Akaroa can be downloaded from the project's website.
Pcap	Allows simulation models to capture and transmit network packets bypassing the operating system's protocol stack. It is not used directly by OMNeT++, but OMNeT++ detects the necessary compiler and linker options for models in case they need it.

The exact names of these packages may differ across distributions.

9.3. Determining Package Names

If you have a distro unrelated to the ones covered in this Installation Guide, you need to figure out what is the established way of installing packages on your system, and what are the names of the packages you need.

9.3.1. Tcl/Tk

Tcl/Tk may be present as separate packages (`tcl` and `tk`), or in one package (`tcltk`). The version number (e.g. 8.5) is usually part of the name in some form (`85`, `8.5`, etc). You will need the development packages, which are usually denoted with the `-dev` or `-devel` name suffix.

Troubleshooting:

If your platform does not have suitable Tcl/Tk packages, you may still use OMNeT++ to run simulations from the command line. To disable the graphical runtime environment use:

```
$ NO_TCL=yes ./configure
```

This will prevent the build system to link with Tcl/Tk libraries. This is required also if you are installing OMNeT++ from a remote terminal session.

By default, the `configure` script expects to find the Tcl/Tk libraries in the standard linker path (without any `-Ldirectory` linker option) and under the standard names (i.e. with the `-ltcl8.4` or `-ltcl84` linker option). If you have them in different places or under different names, you have to edit `configure.user` and explicitly set `TK_LIBS` there (see the Build Options chapter for further details).

If you get the error *no display and DISPLAY environment variable not set*, then you're either not running X (the `wish` command, and thus `./configure` won't work just in the console) or you really need to set the `DISPLAY` variable (`export DISPLAY=:0.0` usually does it).

If you get the error: *Tcl_Init failed: Can't find a usable init.tcl...*

The `TCL_LIBRARY` environment variable should point to the directory which contains `init.tcl`. That is, you probably want to put a line like

```
export TCL_LIBRARY=/usr/lib/tcl8.4
```

into your `~/.bashrc`.

If you still have problems installing Tcl/Tk, we recommend visiting the OMNeT++ site's wiki packages for further troubleshooting tips: http://www.omnetpp.org/pmwiki/index.php?n=Main.TclTkRelatedProblems

9.3.2. BLT

BLT is not a supported package in all Unix distributions. If your distro does not have it, then your can either use Tkenv without BLT (not a big problem), or download and build BLT yourself.

You can get the source tarball, `BLT2.4z.tar.gz` from the Sourceforge BLT project page, http://sourceforge.net/projects/blt.

Unpack the source tgz, change into its directory, then type `./configure`, then `make` and `make install`. (For the last step you'll need root privileges — you may need to contact your sysadmin). If there's any problem, refer to the `INSTALL` file in the package.

You can try your installation by running `bltwish`, or running `wish` and typing

```
% package require BLT
```

in it.

If you don't have root privileges or you get an error during the build, please check the following page: http://www.omnetpp.org/pmwiki/index.php?n=Main.TclTkRelatedProblems

9.3.3. The Java Runtime

You need to install the Sun JRE or OpenJDK. On Unix platforms other than Linux the IDE is not supported so JRE is not required either. We have tested various other Java runtimes (*gcj*, *kaffe*, etc.), and the IDE does not work well with them.

Java version 5.0 (i.e. JRE 1.5) or later is required.

9.3.4. MPI

OMNeT++ is not sensitive to the particular MPI implementation. You may use OpenMPI, or any other standards-compliant MPI package.

9.4. Downloading and Unpacking

Download OMNeT++ from http://omnetpp.org. Make sure you select to download the generic archive, `omnetpp-4.2.2-src.tgz`.

Copy the archive to the directory where you want to install it. This is usually your home directory, `/home/<you>`. Open a terminal, and extract the archive using the following command:

```
$ tar xvfz omnetpp-4.2.2-src.tgz
```

This will create an `omnetpp-4.2.2` subdirectory with the OMNeT++ files in it.

9.5. Environment Variables

In general OMNeT++ requires that its bin directory should be in the PATH. You should add a line something like this to your `.bashrc`:

```
$ export PATH=$PATH:$HOME/omnetpp-4.2.2/bin
```

You may also have to specify the path where shared libraries are loaded from. Use:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/omnetpp-4.2.2/lib
```

If `configure` complains about not finding the Tcl library directory, you may specify it by setting the `TCL_LIBRARY` environment variable.

If you use a shell other than bash, consult the man page of that shell to find out which startup file to edit, and how to set and export variables.

9.6. Configuring and Building OMNeT++

In the top-level OMNeT++ directory, type:

```
$ ./configure
```

The `configure` script detects installed software and configuration of your system. It writes the results into the `Makefile.inc` file, which will be read by the makefiles during the build process.

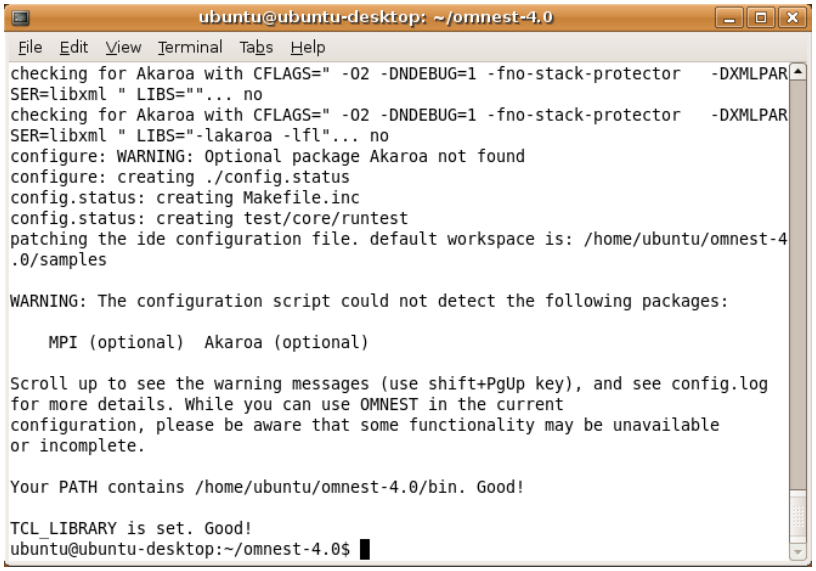


Figure 9.1. Configuring OMNeT++

Normally, the `configure` script needs to be running under the graphical environment (X11) in order to test for `wish`, the Tcl/Tk shell. If you are logged in via an ssh session, or there is some other reason why X is not running, the easiest way to work around the problem is to tell OMNeT++ to build without Tcl/Tk. To do that, use the command

```
$ NO_TCL=1 ./configure
```

instead of plain `./configure`.

If there is an error during `configure`, the output may give hints about what went wrong. Scroll up to see the messages. (Use Shift+PgUp; you may need to increase the scrollback buffer size of the terminal and re-run `./configure`.) The script also writes a very detailed log of its operation into `config.log` to help track down errors. Since `config.log` is very long, it is recommended that you open it in an editor and search for phrases like *error* or the name of the package associated with the problem.

The `configure` script tries to build and run small test programs that are using specific libraries or features of the system. You can check the `config.log` file to see which test program has failed and why. In most cases the problem is that the script cannot figure out the location of a specific library. Specifying the include file or library location

in the `configure.user` file and then re-running the `configure` script usually solves the problem.

When `./configure` has finished, you can compile OMNeT++. Type in the terminal:

```
$ make
```

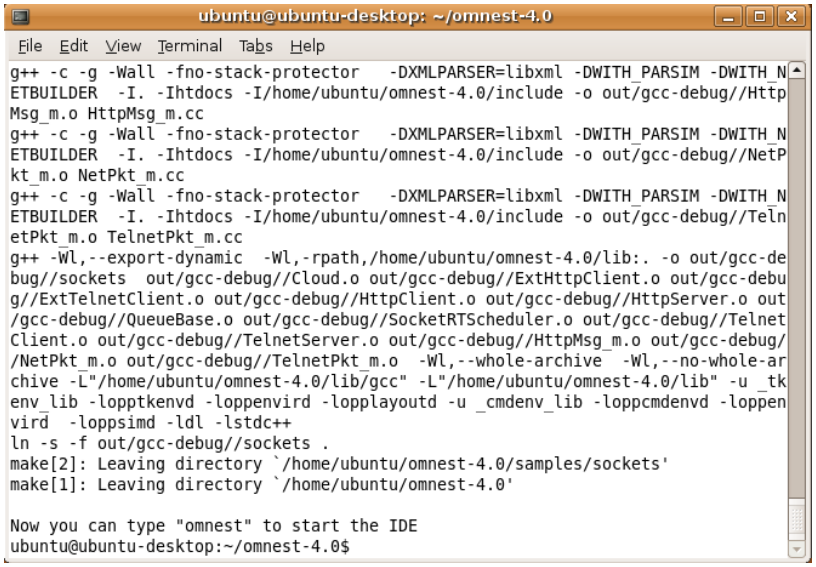


Figure 9.2. Building OMNeT++

To take advantage of multiple processor cores, add the `-j2` option to the `make` command line.

The build process will not write anything outside its directory, so no special privileges are needed.

The make command will seemingly compile everything twice. This is because both debug and optimized versions of the libraries are built. If you only want to build one set of the libraries, specify `MODE=debug` or `MODE=release`:

```
$ make MODE=release
```

9.7. Verifying the Installation

You can now verify that the sample simulations run correctly. For example, the dyna simulation is started by entering the following commands:

```
$ cd samples/dyna
$ ./dyna
```

By default, the samples will run using the Tcl/Tk environment. You should see nice gui windows and dialogs.

9.8. Starting the IDE

The IDE is supported only on Windows, Mac OS X (x86) and Linux (x86,x64).

You can run the IDE by typing the following command in the terminal:

```
$ omnetpp
```

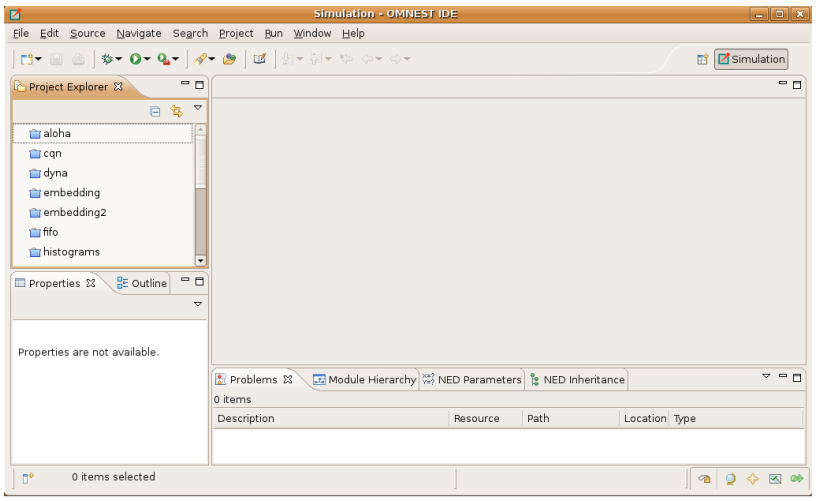


Figure 9.3. The Simulation IDE

If you would like to be able to access the IDE from the application launcher or via a desktop shortcut, run one or both of the commands below:

```
$ make install-menu-item
$ make install-desktop-icon
```

The above commands assume that your system has the `xdg` commands, which most modern distributions do.

9.9. Optional Packages

9.9.1. Akaroa

If you wish to use Akaroa, it must be downloaded, compiled, and installed manually before installing OMNeT++.

As of version 2.7.9, Akaroa only supports Linux and Solaris.

Download Akaroa 2.7.9 from: http://www.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_group/akaroa/download.chtml

Extract it into a temporary directory:

```
$ tar xfz akaroa-2.7.9.tar.gz
```

Configure, build and install the Akaroa library. By default, it will be installed into the `/usr/local/akaroa` directory.

```
$ ./configure
$ make
$ sudo make install
```

Go to the OMNeT++ directory, and (re-)run the `configure` script. Akaroa will be automatically detected if you installed it to the default location.

9.9.2. PCAP

The optional Pcap library allows simulation models to capture and transmit network packets bypassing the operating system's protocol stack. It is not used directly by OMNeT++, but OMNeT++ detects the necessary compiler and linker options for models in case they need it.

Chapter 10. Build Options

10.1. Configure.user Options

The `configure.user` file contains several options that can be used to fine-tune the simulation libraries.

You always need to re-run the `configure` script in the installation root after changing the `configure.user` file.

```
$ ./configure
```

After this step, you have to remove all previous libraries and recompile OMNeT++:

```
$ make cleanall
$ make
```

Options:

`<COMPONENTNAME>_CFLAGS`, `<COMPONENTNAME>_LIBS`	The `configure.user` file contains variables for defining the compile and link options needed by various external libraries. By default, the `configure` command detects these automatically, but you may override the auto detection by specifying the values by hand. (e.g. `<COMP>_CFLAGS=-I/path/to/comp/includedir` and `<COMP>_LIBS=-L/path/to/comp/libdir -lnameoflib`.)
`WITH_PARSIM=no`	Use this variable to explicitly disable parallel simulation support in OMNeT++.
`WITH_NETBUILDER=no`	This option allows you to leave out the NED language parser and the network builder. (This is needed only if you are building your network with C++ API calls and you do not use the built-in NED language parser at all.)
`NO_TCL=yes`	This will prevent the build system to link with Tcl/Tk libraries. Use this option if your platform does not have a suitable Tcl/Tk package and you will run the simulation only in command line mode. (i.e. You want to run OMNeT++ in a remote terminal session.)
`EMBED_TCL_CODE=no`	Tcl/Tk is a script language and the source of the graphical runtime environment is stored as `.tcl` files in the `src/tkenv` directory. By default, these files are not used directly, but are embedded as string literals in the executable file. Setting `EMBED_TCL_CODE=yes` allows you to move the OMNeT++ installation without caring about the location of the `.tcl` files. If you want to make changes to the Tcl code, you better switch off the embedding with the `EMBED_TCL_CODE=no` option. This way you can make changes to the `.tcl` files and see the changes

immediately without recompiling the OMNeT++ libraries.

`CFLAGS_[RELEASE/DEBUG]` — To change the compiler command line options the build process is using, you should specify them in the `CFLAGS_RELEASE` and `CFLAGS_DEBUG` variables. By default, the flags required for debugging or optimization are detected automatically by the `configure` script. If you set them manually, you should specify all options you need. It is recommended to check what options are detected automatically (check the `Makefile.inc` after running `configure` and look for the `CFLAGS_[RELEASE/DEBUG]` variables.) and add/modify those options manually in the `configure.user` file.

`LDFLAGS` — Linker command line options can be explicitly set using this variable. It is recommended to check what options are detected automatically (check the `Makefile.inc` after running `configure` and look for the `LDFLAGS` variable.) and add/modify those options manually in the `configure.user` file.

`SHARED_LIBS` — This variable controls whether the OMNeT++ build process will create static or dynamic libraries. By default, the OMNeT++ runtime is built as shared libraries. If you want to build a single executable from your simulation, specify `SHARED_LIBS=no` in `configure.user` to create static OMNeT++ libraries and then reconfigure (`./configure`) and recompile OMNeT++ (`make cleanall; make`). Once the OMNeT++ static libraries are correctly built, your own project have to be rebuilt, too. You will get a single, statically linked executable, which requires only the NED and INI files to run.

It is important to completely delete the OMNeT++ libraries (`make cleanall`) and then rebuild them, otherwise it cannot be guaranteed that the created simulations are linked against the correct libraries.

The following symbols can be defined for the compiler if you need backward compatibility with some older OMNeT++ 3.x features. They should be specified on the compiler command line using the `-DSYMBOLNAME` syntax. You can add these options to the `CFLAGS_RELEASE` or `CFLAGS_DEBUG` variables (e.g. `CFLAGS_RELEASE='-O2 -DNDEBUG=1 -DSYMBOLNAME'`).

`USE_DOUBLE_SIMTIME` — OMNeT++ 3.x used `double` as the type for simulation time. In OMNeT++ 4.0 and later, the simulation time is a fixed-point number based on a 64-bit integer.. If you want to work with `double` simulation time for some reason (e.g. during porting an OMNeT++ 3.x model), define the `USE_DOUBLE_SIMTIME` symbol for the compiler.

`WITHOUT_CPACKET` — In OMNeT++ 3.x, methods and data related to the modeling of network packets were included in the `cMessage` class. For these parameters, OMNeT++ 4.x has a new class called `cPacket` (derived from cMessage). If you want get back the old behavior (i.e. having a

single `cMessage` class only), define the `WITHOUT_CPACKET` symbol.

10.2. Moving the Installation

When you build OMNeT++ on your machine, several directory names are compiled into the binaries. This makes it easier to set up OMNeT++ in the first place, but if you rename the installation directory or move it to another location in the file system, the built-in paths become invalid and the correct paths have to be supplied via environment variables.

The following environment variables are affected (in addition to `PATH`, which also needs to be adjusted):

`OMNETPP_IMAGE_PATH` — This variable contains the list of directories where Tkenv looks for icons. Set it to point to the `images/` subdirectory of your OMNeT++ installation.

`OMNETPP_TKENV_DIR` — This variable points to the directory that contains the Tcl script parts of Tkenv, which is by default the `src/tkenv/` subdirectory of your OMNeT++ installation. Normally you don't need to set this variable, because the Tkenv shared library contains all Tcl code compiled in as string literals. However, if you compile OMNeT++ with the `EMBED_TCL_CODE=no` setting and then you move the installation, then you need to set `OMNETPP_TKENV_DIR`, otherwise Tkenv won't start.

`LD_LIBRARY_PATH` — This variable contains the list of additional directories where shared libraries are looked for. Initially, `LD_LIBRARY_PATH` is not needed because shared libraries are located via the *rpath* mechanism. When you move the installation, you need to add the `lib/` subdirectory of your OMNeT++ installation to `LD_LIBRARY_PATH`.

On Mac OS X, `DYLD_LIBRARY_PATH` is used instead of `LD_LIBRARY_PATH`. On Windows, the `PATH` variable must contain the directory where shared libraries (DLLs) are present.

10.3. Using Different Compilers

By default, the configure script detects the following compilers automatically in the path:

- Intel compiler (icc, icpc)
- GNU C/C++ (gcc, g++)
- Sun Studio (cc, cxx)
- IBM compiler (xlc, xlC)

If you want to use compilers other than the above ones, you should specify the compiler name in the `CC` and `CXX` variables, and re-run the configuration script.

Different compilers may have different command line options. If you use a compiler other than the default `gcc`, you may have to revise the `CFLAGS_[RELEASE/DEBUG]` and `LDFLAGS` variables.