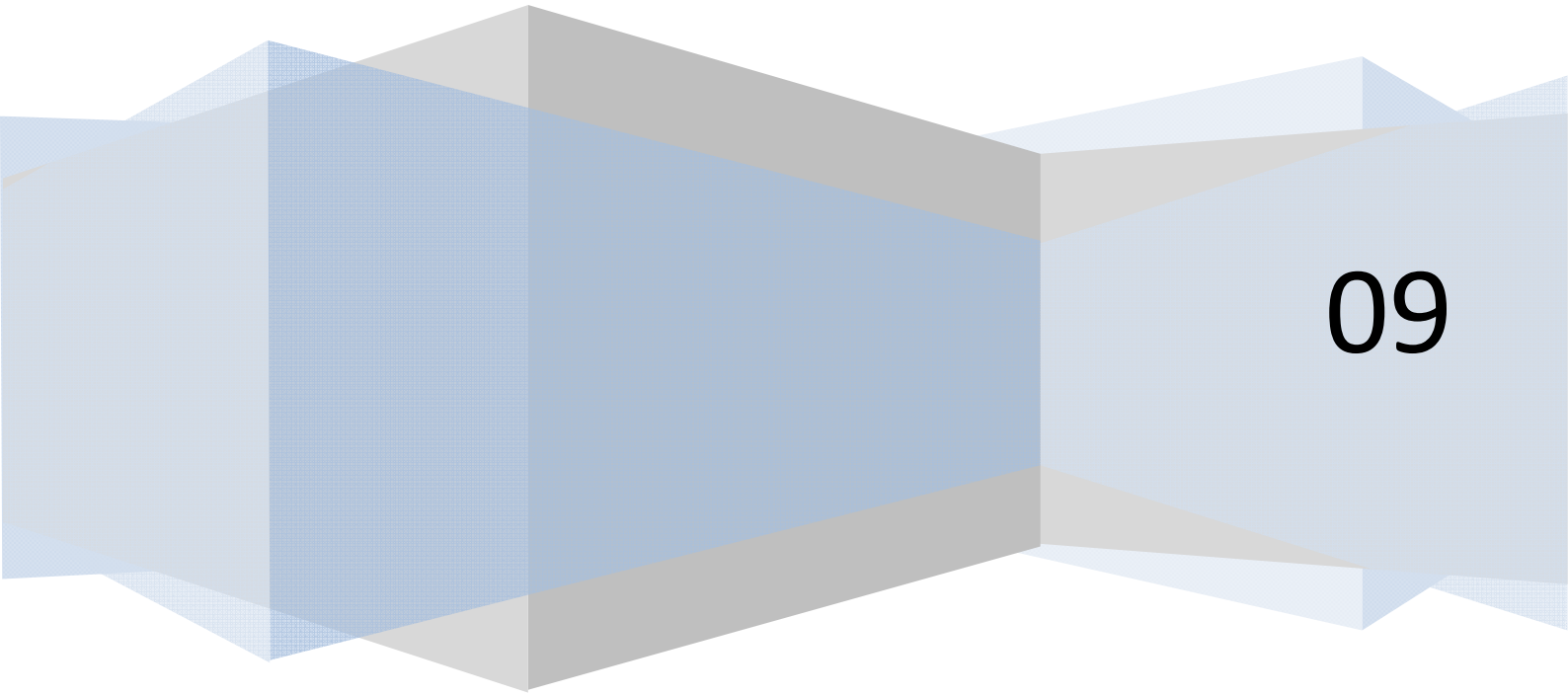
HSR – Hochschule für Technik Rapperswil

Technischer Bericht

Bachelorarbeit TraGiC

Pascal Kesseli & Norbert Schuler



09

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

1.1.1 Ausgangslage

NS-3 ist ein in Forschung und wissenschaftlichen Simulationen eingesetzter Netzwerksimulator, der hoch konfigurier- und anpassbar ist und sich derzeit in Entwicklung befindet. Ein Netzwerk kann mit nahezu beliebiger Komplexität aufgebaut und vernetzt werden, und durch die hohe Flexibilität und Modularisierung von NS-3 können verschiedenste Geräte über alle möglichen Kanäle miteinander kommunizieren.

1.1.2 Problemstellung

Was NS-3 fehlt, sind unter anderem Implementierungen von höheren Protokollen und eine verteilungsbasierte Verkehrsgenerierung. Bisher wurden nur Protokolle auf den unteren OSI-Schichten implementiert, bis hin zu TCP und UDP. Eine Referenzimplementierung eines höheren Protokolls würde die Arbeit der Entwickler entscheidend vereinfachen und gleichzeitig generelle Fragestellungen dieses Problems beantworten. Ebenfalls nötig für komplexe Simulationen ist ein flexibler Verkehrsgenerator, der zwischen beliebigen Kommunikationspartnern Verbindungen aufbaut, parametrisierbare Pakete erstellt und versendet.

1.1.3 Ziel der Arbeit

Als Kombination der zwei Hauptprobleme wird ein Verkehrsgenerator implementiert, welcher Streaming mithilfe des RTP Protokolls simulieren soll, in diesem konkreten Falle Voice-over-IP Verkehr aufgrund von realistischen Verkehrsmodellen und davon hergeleiteten Verteilungsfunktionen.

1.2 Der NS-3 Simulator

NS-3 ist ein eventdiskreter Netzwerksimulator, welcher hauptsächlich in Forschung und wissenschaftlichen Simulationen Verwendung findet. Die Software wird unter der GNU GPL Lizenz entwickelt und ist der Nachfolger von NS2.

Um die Simulation komplexer Netzwerke zu ermöglichen basiert NS-3 auf einer verständlichen und streng hierarchisch aufgebauten Struktur. Einzelne Kommunikationspartner (Router, PCs, in NS-3 generell „Nodes“) können in Gruppen zusammengefasst werden („Node Container“). Auf Nodes oder Node Containern können wiederum Schnittstellen („Net Device“) eingebaut werden, in denen die Adressierung – meist IP-Adressen – gespeichert ist. Im Falle eines Node Containers teilen sich die enthaltenen Nodes einen gewissen Adressbereich und befinden sich somit im selben Subnetzwerk. Die Kommunikation in NS-3 wird über logische Kanäle geregelt, welche physische Umgebungen simulieren können, inklusive Jitter und Verzögerungen.
Um Daten zu verschicken, benötigt man neben der aufgebauten Infrastruktur sogenannte Applications, welche auf den Nodes installiert werden können. Applications sind in ihrer Implementierung sehr frei und übernehmen die Simulationsaufgaben im Netzwerk. Der schlussendliche Output von NS-3 sind PCAP Files, welche sämtliche ein- und ausgehenden Pakete an je einer Schnittstelle beinhalten. Diese Files können zum Beispiel von Wireshark[1] gelesen und analysiert werden. Eine Alternative sind Rohdaten-Files, die dasselbe wie PCAP Files, aber in veränderter Form ausgeben (Text Output).

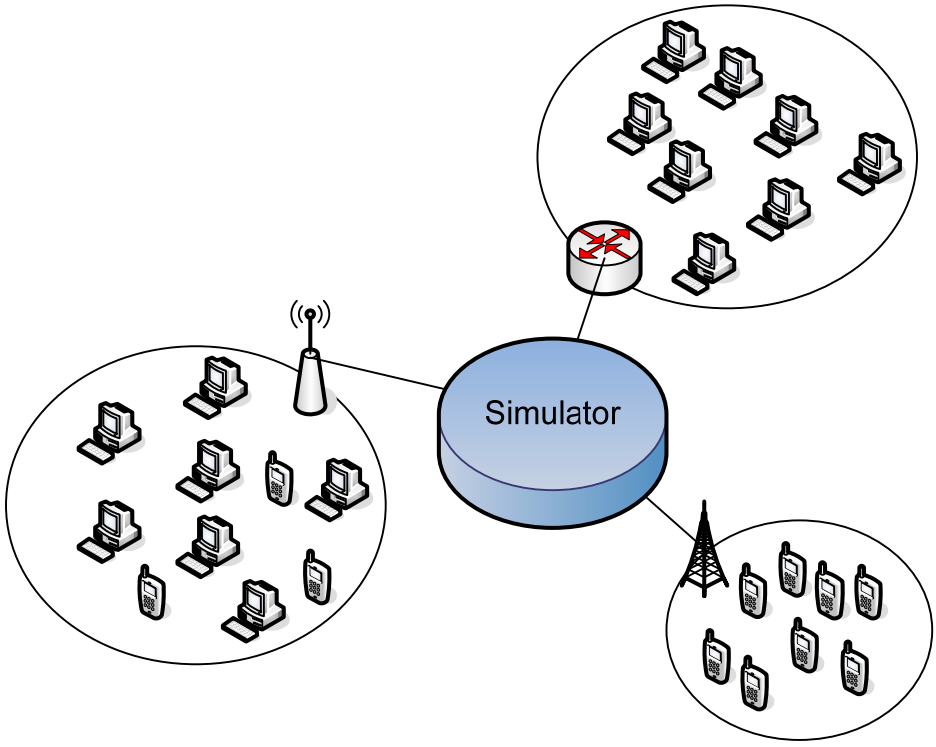


Abbildung 1: Beispielaufbau eines Netzwerkes mit NS-3

Der NS-3-Simulator kann beispielsweise, wie in Abbildung 1 dargestellt, ein beliebiges Netz aus Mobile Devices, Computern und anderen Geräten mit jeder Art von Schnittstelle einbinden. Durch

[1] Programm zur Analyse von Netzwerk-Kommunikationsverbindungen, http://www.wireshark.org/

die hohe Modularität des NS-3 Designs kann der Programmierer neben bereits vorgefertigten Kanälen wie Luft oder simple Ethernet-Kabel auch eine eigene Variante verwenden.

Mit diesen Möglichkeiten kann NS-3 fast jeden beliebigen Netzwerkaufbau simulieren, verschiedene Geräte zusammenfassen und unterschiedlichste Szenarien implementieren, verändern und testen.

1.3 Die NS-3 Simulationsumgebung

NS-3 erlaubt es dem Anwender, eine komplette und realitätsgetreue Netzwerkumgebung aufzubauen. Die zentralen Punkte der Simulation sind dabei die Nodes, welche ein physisches Gerät repräsentieren. Nodes können untereinander über einen Channel verbunden werden und mithilfe von Applications darüber kommunizieren. Dieser Aufbau kann schematisch wie folgt dargestellt werden:

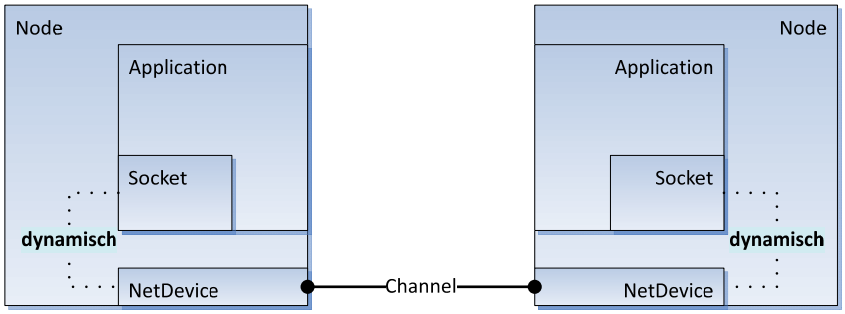


Abbildung 2: Zwei verbundene Nodes in NS-3

Innerhalb der Architektur von NS-3 sieht dies als Klassendiagramm dargestellt so aus:

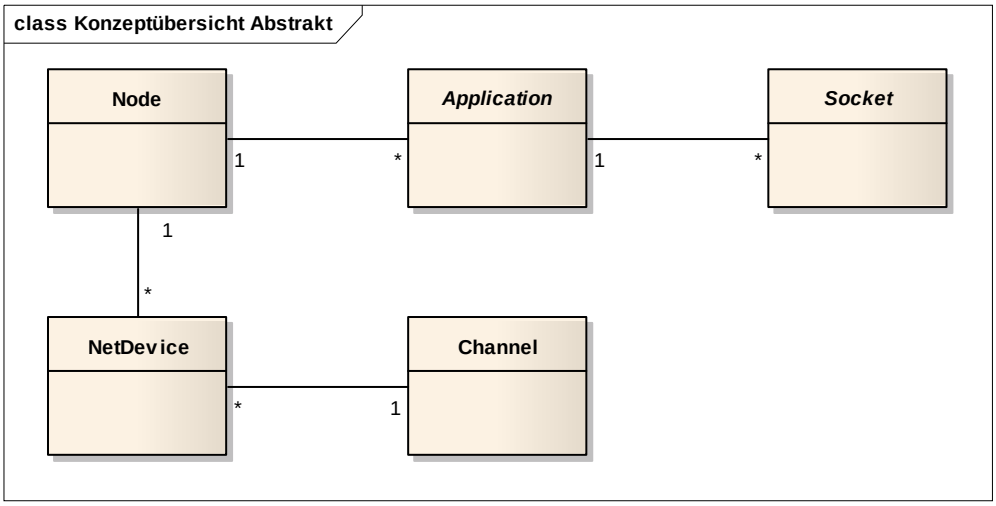


Abbildung 3: Klassendiagramm NS-3, Node

Erklärung der Begriffe:

- Node
  - repräsentiert ein Gerät, das mit dem Netzwerk verbunden werden soll
  - Beispiele für Nodes sind Computer, Handys, aber auch Router oder Server
- NetDevice
  - verbindet den Node mit einem gegebenen Netzwerkkanal

    - Beispiele sind Netzwerkkarten, Antennen
- Channel
    - ist ein physikalischer Verbindungskanal zwischen NetDevices
    - kann verschiedene Medien oder Verbindungen repräsentieren
        - Kabel
        - Funkverbindungen
    - repräsentiert aber auch komplexere Umgebungen
        - z.B. CSMA-Channel, ein Kanal mit Carrier Sense Multiple Access Eigenschaften
- Socket
    - implementiert eine aktive Verbindung zwischen zwei Nodes
    - verwendet Eigenschaften des Nodes (wie z.B. seine NetDevices), um Verbindung zu realisieren
    - implementiert meist ein spezifisches Protokoll (z.B. TcpSocket, UdpSocket)
    - verwendet Header und andere Hilfsklassen zur Implementierung des Protokolls
- Application
    - stellt Anwenderapplikation dar
    - verwendet durch andere Komponenten hergestellte Netzwerkverbindung (meist via Socket)
    - wird durch Anwender implementiert, um ein gewünschtes Verhalten zu simulieren (Server, Clients, Router, …)

1.3.1 Parametersystem

In der obigen Beschreibung der NS-3 Netzwerkarchitektur wird kurz die dynamische Verbindung zwischen Socket und NetDevice angesprochen. Dynamisch deshalb, weil eine Socket-Implementierung in NS-3 meist nicht über eine direkte Verbindung zu seinem NetDevice oder sogar dem zugehörigen Channel verfügt (obwohl das einem NS-3 Entwickler durchaus freistehen würde). Stattdessen verfügt jedes Objekt in NS-3 über folgende Methoden:

- AggregateObject
    - Weist dem Objekt eine Referenz auf ein beliebiges anderes Objekt zu.
- GetObject<T>
    - Gibt eine Referenz des verlangten Typs zurück, welche zuvor mit AggregateObject assoziiert wurde oder
    - konvertiert das Objekt selbst zum verlangten Typ, wenn es das entsprechende Interface implementiert

Die zuvor beschriebene dynamische Bindung ist also die Möglichkeit in NS-3, beliebigen Objekten Referenzen auf andere Objekte zuzuweisen und jederzeit im Programm auf die so verbundenen Objekte zuzugreifen. Auf diese Weise werden zentrale Klassen wie Sockets oder Channels gleichermassen miteinander verbunden wie Protokoll- und Hilfsklassen. Ein konkreter Fall in NS-3 ist z.B.

- Node
    - UdpSocket
        - UdpSocketImpl
            - UdpL4Protocol

        - Ipv4Protocol
    - NetDevice
        - Queue
        - Channel
    - …

Ein UdpSocket kennt also seinen Node, auf dessen NetDevice, Queue und Channel er ohne eine direkte Referenz dank GetObject<T> zugreifen kann. Diese Verknüpfungen korrekt zu erstellen ist oftmals sehr aufwändig und fehleranfällig, da die Objekte zwar beliebig verknüpft werden können, die meisten Protokolle und Funktionen ihr Vorhandensein aber an gewissen Punkten voraussetzen. Aus diesem Grund stellt NS-3 diverse Helper-Klassen zur Verfügung, welche die korrekte Installation und Verknüpfung der betreffenden Objekte sicherstellen.[2]

1.3.2 Simulationszeit

Ein wichtiges Konzept, das bei der Programmierung mit der NS-3 Simulationsumgebung stets beachtet werden muss, ist, dass es sich um keine Echtzeit-Simulation handelt. Die Simulation läuft völlig unabhängig von der realen Zeit, sodass beispielsweise mehrere Tage an Simulationsdaten innert weniger Minuten generiert werden können. Diese Zeitabstraktion muss allerdings auch bei der Entwicklung beachtet werden und führt zu folgenden Überlegungen:

- Scheduling
    - Die einzige Möglichkeit, ein Ereignis oder eine Aktion auszuführen, ist, es beim Scheduler des Simulators zu registrieren.
    - Um ein Ereignis zu registrieren, muss eine Ausführungszeit mitgegeben werden. Möchte man also eine Aktion ausführen, muss stets bekannt sein, zu welchem sie ausgeführt werden soll.
    - Im Normalfall läuft eine Simulation ohne Unterbruch bis zu ihrem Ende durch, ohne dass man weitere Ereignisse planen könnte. Während der Simulation können allerdings die vom Scheduler aufgerufenen Methoden weitere Ereignisse registrieren.
- Parallelität
    - Während der Simulation existiert keine Parallelität in Form von Threads oder ähnlichen Mechanismen. Zu jedem Zeitpunkt ist nur ein Simulator vorhanden, der zuvor geplante Events sequentiell am Stück abarbeitet.
    - Sollen mehrere Events gleichzeitig ausgeführt werden, so lassen sich diese beim Simulator für denselben Zeitpunkt registrieren. Der Simulator arbeitet diese dann sequentiell ab, ohne dass er die Simulationszeit weiterführt.
- Genauigkeit
    - NS-3 erlaubt das Planen von Ereignissen mit bis zu einer Femtosekunde Genauigkeit. Dabei muss allerdings beachtet werden, dass mit derart kleinen Einheiten sehr schnell sehr grosse Zahlenwerte entstehen können. Die NS-3-Zeitstrukturen bieten in diesen Fällen keinen Schutz vor Zahlenbereich-Überlauffehlern.

[2] NS-3 Tutorial, ns-developers@isi.edu, http://www.nsnam.org/docs/release/tutorial.pdf

1.4 Problemstellung, Ziele

Für NS-3 existieren bisher noch keine Verkehrsgeneratoren (oder nur Ansätze dazu), die jedoch für Tests und Simulationen dringend vonnöten wären. Aufwendige Simulationen werden dadurch erschwert. Weiterhin existiert noch kein höheres Protokoll, das auf TCP beziehungsweise UDP aufsetzt. Dies rührt daher, dass NS-3 zum Zeitpunkt dieser Arbeit noch in Entwicklung ist und erst die Protokolle der unteren OSI-Schichten modelliert wurden.

Eine denkbare Anwendung und Kombinierung der beiden Probleme ist ein Verkehrsgenerator, welcher Voice-over-IP Verkehr simuliert. VoIP Verkehr benützt meist RTP, ein auf UDP aufsetzendes Streaming Protokoll. Durch die Implementierung einer solchen Applikation ermöglicht man dem NS-3 Team, auf Basis dieser Arbeit weitere Generatoren zu entwickeln und grössere Netzwerke mit VoIP Verkehr zu simulieren.

Das Projekt TraGiC („Traffic Generator – Inherently Consequential“) soll diese Lücke schliessen. Ein Konzept für den Versuchsaufbau für die VoIP Netzwerksimulation beinhaltet ein Netzwerklayout mit verschiedenen Gruppen, welche Subnetze mit VoIP-Kommunikationspartnern darstellen. In diesen Gruppen befinden sich beliebig viele Nodes (Teilnehmer), welche jeweils den Traffic Generator als NS-3-Application installiert haben, um die Kommunikation zu simulieren. Der effektive Verkehr, seine Ausmasse und andere Parameter werden mit NS-3 konform vom User vorgegeben. Damit erzeugt TraGiC Pakete, die dem VoIP Standard RTP entsprechen, und NS-3 generiert daraufhin die üblichen PCAP Files mit den korrekten Daten.

1.5 Aufgabenbereich TraGiC

Wie im vorherigen Kapitel erwähnt, liegt es am Benutzer, eine entsprechende Applikation zu programmieren, die das Verhalten implementiert, das er simulieren möchte. Eines der häufigsten Simulationsszenarien sind Belastungs- und Verkehrstests mit Verkehrsgeneratoren. Dabei stehen der Payload und die Applikationsdaten einer Übertragung im Hintergrund, während einfach eine gewünschte Auslastung über eine vorgegebene Zeit aufrecht erhalten werden muss. Dies dient dazu, Komponenten und andere Applikationen auf ihre Robustheit zu prüfen.

NS-3 stellt keine solche Vorimplementierung eines Verkehrsgenerators bereit und momentan sind auch noch keine Protokolle höherer Layer als TCP oder UDP auf Layer 4 verfügbar. Um also eine Simulation, wie in der Aufgabenstellung beschrieben, mit einem Traffic-Generator, der RTP-Verkehr generiert, erstellen zu können, muss das obenstehende Diagramm wie folgt erweitert werden:

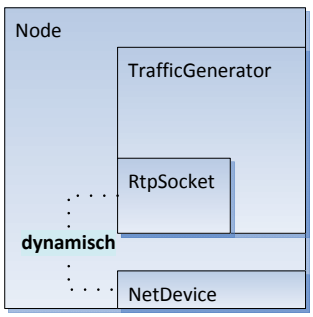


Abbildung 4: Spezialisierte TraGiC Node

In der Klassenübersicht müssen die abstrakten Klassen erweitert werden:

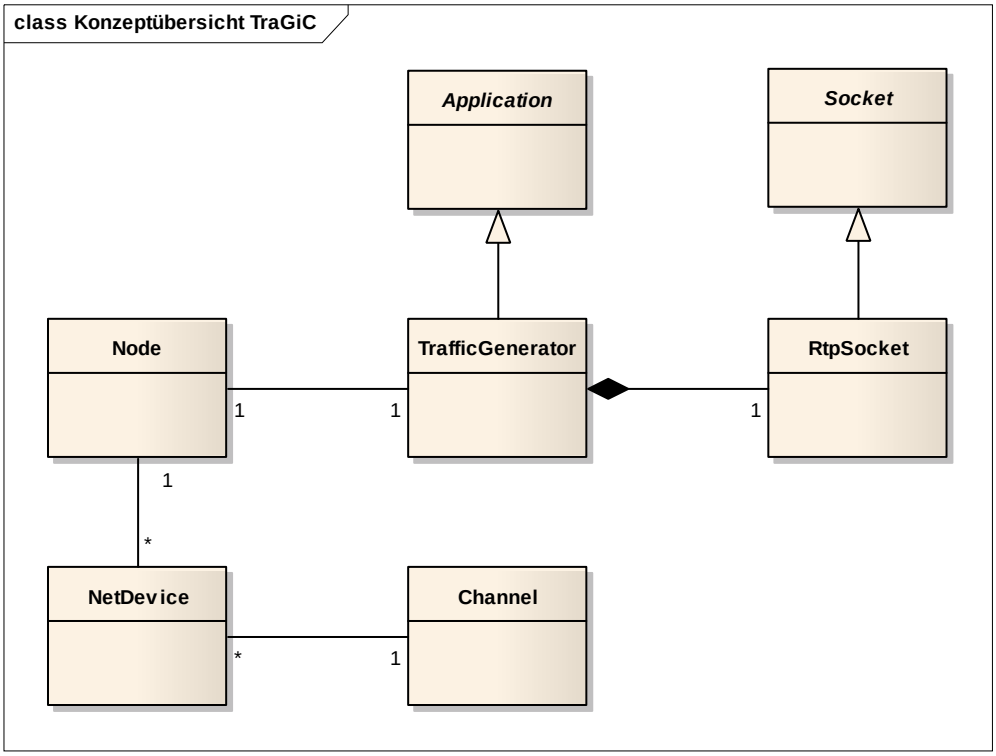


Abbildung 5: TraGiC-Komponenten in NS-3

Erklärung der Begriffe:

- TrafficGenerator
  - eine spezielle NS-3-Applikation mit dem Ziel, Verkehr verschiedener Protokolle aufgrund unterschiedlicher Parameter zu erzeugen
  - zu erreichende Lasten und Intervalle sollen konfigurierbar sein
- RtpSocket
  - implementiert das RTP-Protokoll
  - erlaubt das senden beliebiger Payloads (SDUs)
  - verpackt erhaltene Payloads zu RTP-PDUs

Wenn man dieses Diagramm mit dem ursprünglichen NS-3-Konzeptdiagramm vergleicht, haben sich neben den TraGiC-Klassen auch die Multiplizitäten geändert. Dies deshalb, weil für TraGiC andere Regeln gültig gemacht werden müssen: Zwar kann eine Node mehrere Application-Objekte beinhalten, für die korrekte Ausführung des Traffic Generators aber nur eine Instanz davon. Ähnlich verhält es sich mit dem Socket: Ein TrafficGenerator-Objekt aggregiert jeweils genau einen Socket, obwohl mit NS-3 beliebig viele möglich wären.

1.6 Featureliste / Funktionen

1.6.1 Erforderlich

1.6.1.1 Traffic Generator

Als Hauptfeature soll es mit TraGiC möglich sein, realistischen Verkehr aufgrund verschiedener statistischer Datenquellen und Konfigurationsangaben zu simulieren. Dabei soll TraGiC als

Applikation im simulierten NS-3-Netzwerk laufen und den Einfluss von verschiedenen Verkehrssituationen auf Leistung und Zuverlässigkeit der Netzwerke aufzeigen.
Der zu implementierende Traffic Generator soll modular und flexibel erweiterbar sein, um verschiedene Protokolle zu unterstützen. Zudem soll es möglich sein, die grundlegenden Funktionen des Traffic Generators um spezielle Implementierungen (z.B. protokollspezifisches Verhalten in der Applikation) zu erweitern.

1.6.1.2 RTP Implementierung

Um den Voice-over-IP Verkehr in NS-3 zu simulieren, soll eine Implementierung des RTP Protokolls entwickelt werden. Diese soll es ermöglichen, realistische RTP VoIP-Pakete inklusive Header und Sprach-Payload über das NS-3 Netzwerk zu versenden und bildet damit den Vorreiter für Protokolle höherer Layers in NS-3. Gleichzeitig soll eine spezialisierte Version des Traffic Generators Verkehr über RTP simulieren, um auch hiervon eine Referenzimplementierung für das NS-3 Team bereitzustellen.

1.6.1.3 Mathematische Verteilungsfunktionen zur Verkehrsgenerierung

TraGiC soll seinen Verkehr über die Angabe von Zufallsvariablen mithilfe mathematischer Verteilungsfunktionen und Seeds generieren können. Dazu wird ein Modul benötigt, welches nach diesen Verteilungen starke Zufallszahlen erstellen kann. NS-3 bietet dank seinem Bedürfnis nach Zufall (Störungen in Leitungen, Interferenzen) ein ausgereiftes Mathematikmodul, welches in TraGiC wiederverwertet werden kann. Somit wird nur noch eine Schnittstelle benötigt.

1.6.2 Optional

1.6.2.1 Erweiterte Konfigurationsmöglichkeiten für RTP

Nebst der zuvor beschriebenen Konfiguration des Verkehrs soll es TraGiC erlauben, die verschiedenen Parameter der RTP Implementierung für eine Simulation vorzugeben. Namentlich soll es möglich sein, folgende Einstellungen vorzunehmen:

- Auswahl des Codecs
  - Vorgegebene Payload Grösse
  - Vorgegebene Paketrate
- Verkehrstabelle
  - Verkehrsaufkommen zwischen verschiedenen Gruppen
- Verteilung für Abstände zwischen zwei Gesprächsanfängen
- Anzahl Teilnehmer pro Gruppe
- Durchschnittliche Dauer eines Gesprächs

2 Analyse

Nachfolgend werden die einzelnen Problembereiche, die durch obige Aufgabenstellung vorgegeben sind, analysiert und mögliche Lösungsvarianten gesucht.

2.1 Verkehrsgenerator

2.1.1 Allgemein

Konzeptionell wird der Verkehrsgenerator als NS-3-Applikation modelliert, deren Grundfunktion lediglich das Versenden von Paketen ist. In welchen Massen und Intervallen dies geschehen soll und welche Protokolle dabei verwendet werden sollen, muss frei konfigurierbar sein. Daraus ergeben sich folgende Schwerpunkte, die in den nächsten Kapiteln bearbeitet werden:

- Parametrisierung
- Unterstützung unterschiedlicher Protokolle

2.1.2 Parametrisierung

Der Benutzer der Applikation muss die Möglichkeit haben, dem Traffic-Generator folgende Verkehrseigenschaften vorzugeben:

- Paketraten
- Paketgrössen
- Zieladresse
- Zeitabstände zwischen Paketen
- Gesprächsdauer
- Abstand zwischen zwei Gesprächen

Diese Werte sollen vom Benutzer beliebig gesetzt werden können, und zwar für jedes einzelne versendete Paket. Es muss somit eine Schnittstelle bereitgestellt werden, die dem Benutzer diese Parametrisierung im Programm ermöglicht. Diese abstrakte Schnittstelle wird in der Architekturdokumentation dieses Berichts 1:1 als Software-Interface umgesetzt. Wichtig ist dabei, dass der Traffic-Generator für jedes zu versendende Paket die korrekten Werte abfragen kann. Der Benutzer, der diese Schnittstelle nutzt, muss also nicht nur einzelne Werte, sondern stets eine Reihe von Werten für die jeweiligen Parameter zur Verfügung stellen. Ob diese Reihe irgendwelchen Mustern oder Gesetzen folgt, bleibt ganz dem Benutzer überlassen. Somit kann er beliebige Verkehrsmuster durch die Spezifizierung entsprechender Wertereihen erzeugen.

2.1.2.1 Konkrete Implementierung durch Zufallsvariablen

Eine spezielle Nutzung dieses Parameterinterfaces sei an dieser Stelle noch explizit erwähnt: Bei der Simulation von Netzwerkverkehr ist es ein gängiges Szenario, aufgrund von Erfahrungen und Messungen eine statistische Beschreibung des erwarteten Verkehrs zu erstellen. Das bedeutet, dass meist eine zu erzeugende Lastverteilung bekannt ist, die sich am einfachsten als Verteilungsfunktion bzw. Zufallsvariable darstellen liesse. Eine sehr häufig genutzte Implementierung der vorhin beschriebenen Schnittstelle ist deshalb das Generieren der benötigten Parameter-Reihen aufgrund einer statistischen Verteilung.

Dieser sehr häufige Use Case ist im fertigen Produkt schon fest vorgesehen, sodass der Benutzer den Traffic-Generator auch allein durch die Angabe der gewünschten Verteilungen pro Parameter konfigurieren kann.

2.1.2.2 Verkehrsverteilung

Wie in der Einleitung beschrieben, soll Verkehr zwischen und innerhalb von verschiedenen Subnetzen simuliert werden. Eine gute Möglichkeit, den aufkommenden Verkehr zu beschreiben, ist eine vom User vorgegebene Verkehrsverteilungstabelle.

Als Beispiel werden drei Netze A, B und C definiert. Die eingegebenen Werte sind jedoch nicht absolut, sondern werden pro Zeile auf 1 normiert, um die Wahrscheinlichkeit eines Streams in dieses Netzwerk zu erhalten. In der folgenden Tabelle könnte somit in der zweiten Zeile beispielsweise auch 12-0-12 stehen, um dieselbe Verteilung zu erhalten: B sendet nach A und nach C jeweils mit der Hälfte der möglichen Kapazität.

	A	B	C
A	3.6	6.4	10
B	0.5	0	0.5
C	3	3	3

Tabelle 1: Beispiel einer Verteilungstabelle

Die mögliche Kapazität eines Netzwerkes ist durch die Anzahl Nodes darin gegeben.

2.1.3 Unterstützung unterschiedlicher Protokolle

Aus den Anforderungen an den Verkehrsgenerator geht hervor, dass er beliebige Netzwerkprotokolle für die Simulation unterstützen muss. Das bedeutet, dass die erstellten Protokoll-PDUs unabhängig von den vom Traffic-Generator erzeugten Dummy-Daten SDUs erstellt werden sollen. Für diese Abstraktion bietet die Architektur zwei Ansatzpunkte:

- Packet
  In NS-3 ist jede SDU ein Packet-Objekt, egal ob die simulierten Protokolle andere Begriffe dafür verwenden (so werden z.B. auch Pakete anstatt Frames über Ethernet gesendet). Generiert der Traffic-Generator also NS-3-Packets, so generiert er SDUs, die für jedes in NS-3 simulierte Protokoll verwendet werden können.
- Socket
  Wie auch in echten Netzwerken ist es nicht die Aufgabe der Benutzerapplikation, SDUs in PDUs zu verpacken. Dazu dient in NS-3 eine bei allen Protokollen identische Schnittstelle: Der Socket. Wenn der Verkehrsgenerator die Socket-Schnittstelle nutzt, kann er durch einfaches Austauschen des Socket-Objekts Verkehr über beliebige Protokolle generieren.

2.1.4 Paketgenerierung via NS-3

Wie in der Einführung beschrieben, gelten für die NS-3 Simulationsumgebung besondere Bedingungen bezüglich dem Planen und Ausführen von Ereignissen. Ist eine Simulation einmal gestartet, arbeitet sie atomar alle geplanten Ereignisse bis zum Simulationsende ab. Daraus folgt:

- Für eine Simulation müssen entweder alle Ereignisse bereits vor Simulationsbeginn mit ihrem zugehörigen Ausführungszeitpunkt bekannt sein, oder
- Die beim Scheduler registrierten Methoden müssen bei ihrer Ausführung selbst neue Methoden registrieren.

Letztere Variante ist insbesondere dann interessant, wenn die Simulationsdauer nicht fest vorgegeben ist, sondern die Simulation erst durch Beenden des Simulators selbst (und nicht der

Applikationen) abgeschlossen wird. Diese Variante ist ausserdem dahingehend flexibler, dass sie auch das Starten von Aktionen aufgrund von Simulationsereignissen oder -resultaten ermöglicht.

Beide oben beschriebenen Eigenschaften sind Gründe dafür, den Traffic-Generator nach der zweiten Variante zu implementieren. Da der TraGiC-Verkehrsgenerator allerdings nur eine einzige, für die Simulation wesentliche Funktion beinhaltet – nämlich das Versenden eines Pakets – entsteht daraus folgendes, repetitives Aufrufschema:

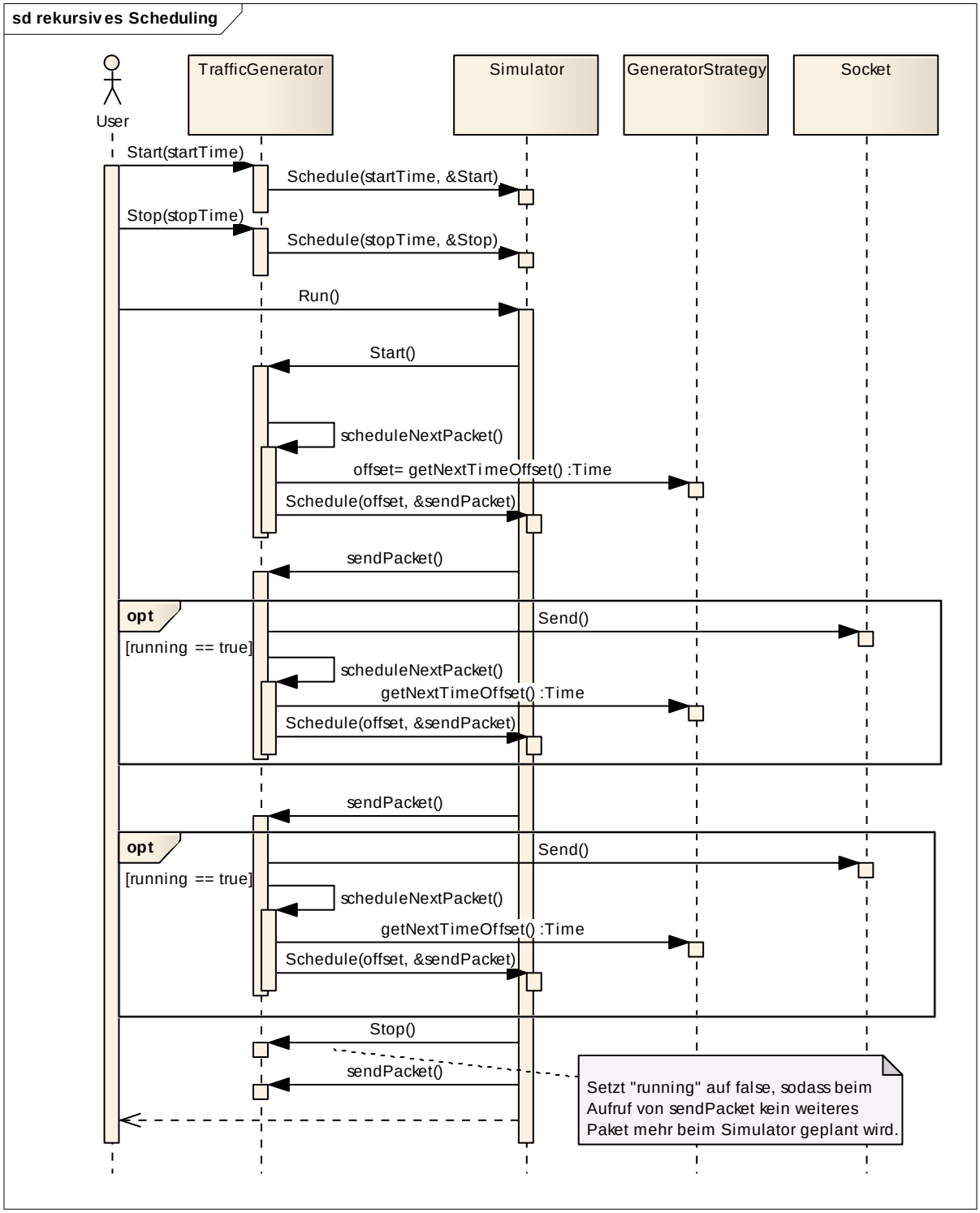


Abbildung 6: Repetitives, selbstaufrufendes Paketscheduling

Obige Grafik zeigt deutlich auf, dass, solange die Simulation abläuft, die Methode sendPacket des TrafficGenerators immer wieder repetitiv vom Simulator aufgerufen wird. Ihren nächsten Aufruf plant die sendPacket Methode dabei selbst anhand der von der GeneratorStrategy bereitgestellten Parameter und meldet diesen beim Scheduler an. Daraus entsteht ein der Rekursion ähnliches Aufrufschema, welches erst durch den zuvor geschedulten Aufruf von Stop() beendet wird.

2.2 RTP

Obwohl für zahlreiche Protokolle und Applikationen anwendbar, ist das primäre Ziel von TraGiC, Verkehrssituationen im Voice-over-IP Bereich zu simulieren. Um dies zu ermöglichen, wird im Rahmen dieses Projekts nebst dem eigentlichen, modularen Traffic-Generator auch eine konkrete Implementierung desselbigen für das RTP Protokoll erstellt. RTP stellt in der heutigen IP-Kommunikation de facto den Standard für die Übertragung von Sprach- und Videodaten dar.

Um RTP in NS-3 zu simulieren, müssen zuerst einige Überlegungen angestellt werden, die in den folgenden Kapiteln behandelt werden.

2.2.1 RTP im OSI Modell

Das RTP-Protokoll setzt in seiner Funktion direkt auf dem UDP/IP Protokoll auf, welches in NS-3 bereits komplett implementiert ist. Wie im OSI-Modell üblich, kann die RTP-PDU als einfache SDU an die UDP-Schicht in NS-3 weitergegeben werden. Das bedeutet, dass sich TraGiC nur mit dem RTP-Protokoll selbst, welches zwischen OSI-Layer 4 und 5 agiert, auseinandersetzen muss.

Transport	**RTP** UDP, TCP, SPX, SCTP
Network	IP, IPX, ICMP
Data Link Physical	Ethernet, Token Ring, FDDI

Tabelle 2: RTP im OSI-Modell[3]

Formell gesehen ist RTP laut RFC ein Layer 4 Transportprotokoll, setzt aber in der Praxis oftmals auf einem vorhandenen Transportprotokoll auf und kann deshalb als Layer 4.5-Protokoll angesehen werden. Es ist unabhängig von den darunterliegenden Protokollen, wird aber meist zusammen mit UDP und IP verwendet.

Als reiner Datencontainer ist RTP zuständig für die Übertragung von Multimediadaten, enthält aber keine Funktionen zur Flusssteuerung oder Fehlerkontrolle und bietet keinen garantierten Quality of Service. Im Prinzip beruht RTP darauf, dass der Sender Audio- und Videodaten in RTP-Pakete verpackt und mit Zusatzinformationen versieht, die von der Empfänger-Applikation genutzt werden können. Unter anderem sind dies Codec-Daten (Payload Typ, Codierung, Paketrate) und Sequenznummern/Timestamps zur Kennzeichnung von zusammengehörenden Audio- oder Videofragmente. Beim Streaming selbst werden Audio- und Videodaten meist getrennt behandelt und auch an verschiedene Ports verschickt.

RTP wird häufig durch die beiden Protokolle RTCP (Real-Time Transport Control Protocol) und RTSP (Real-Time Streaming Protocol) ergänzt. Das RTCP hat die Aufgabe, die Teilnehmer und den Quality

[3] http://de.wikipedia.org/wiki/Real-time_Transport_Protocol, 01.04.09

of Service einer RTP-Session zu überwachen. Dazu tauschen alle RTP-Sender und -Empfänger periodisch Statusinformationen aus, welche dazu genutzt werden können, die Datenrate des Senders anzupassen und eine bessere Übertragung zu ermöglichen.
RTSP dient zur Steuerung des Streams. Es ähnelt in Aufbau und Verhalten HTTP und benutzt ebenfalls eine eindeutige Adresse der Form „rtsp://ita.hsr.ch:554/stream“. Dabei kennzeichnet „rtsp“, dass RTSP-Befehle über TCP ausgetauscht werden sollen, „ita.hsr.ch“ ist der Hostname und 554 der Standardport für das Protokoll. Im Gegensatz zu HTTP ist RTSP allerdings bidirektional, das heisst sowohl Client als auch Server können Anfragen absetzen. RTSP wird häufig auch als „Fernbedienung über Internet“ bezeichnet und unterstützt sowohl Uni- als auch Multicast.[4]

Für den Voice-over-IP-Verkehrsgenerator reicht es in diesem ersten Schritt aus, "reine" RTP-Pakete zu versenden, da noch keine echten VoIP-Gespräche aufgesetzt werden, weshalb sich diese Arbeit auf RTP beschränkt und auf eine gesamtumfassende Implementierung eines Streamingsystems verzichtet wird. Dies wäre vielmehr eine mögliche Erweiterung, welche auf den Ergebnissen von TraGiC aufsetzt.

2.2.2 RTP-Header

Wie bei den meisten Protokollen liegt auch bei RTP der Schwerpunkt der Implementierung auf dem Header des Protokolls. Sämtliche Kommunikations- und Flusskontrollparameter sind darin enthalten:

<table>
<tr><th colspan="8">Byte 0</th><th colspan="8">Byte 1</th><th colspan="8">Byte 2</th><th colspan="8">Byte 3</th></tr>
<tr><td>Bit 0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>Bit 0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>Bit 0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>Bit 0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td colspan="4">V=2</td><td>P</td><td>X</td><td colspan="2">CC</td><td>M</td><td colspan="7">PT</td><td colspan="16">sequence number</td></tr>
<tr><td colspan="32">timestamp (in sample rate units)</td></tr>
<tr><td colspan="32">synchronization source (SSRC) identifier</td></tr>
<tr><td colspan="32">contributing source (CSRC) identifiers (optional)</td></tr>
<tr><td colspan="32">Header Extension (optional)</td></tr>
</table>

Tabelle 3: RTP-Header Aufbau[5]

Folgende Beschreibung aus dem deutschen Wikipedia-Artikel über RTP[6] fasst den RFC-Standard des RTP-Headers ausreichend zusammen:

- *Version (V), 2 Bit*
  Versionsstand des RTP-Protokolls
- *Padding (P), 1 Bit*
  Das Füll-Bit ist gesetzt, wenn ein oder mehrere Füll-Oktets am Ende des Pakets angehängt sind, die nicht zum eigentlichen Dateninhalt (Payload) gehören. Das letzte Füll-Oktet gibt die Anzahl der hinzugefügten Füll-Oktets an. Füll-Oktets werden nur dann benötigt, wenn nachfolgende Protokolle eine vorgegebene Blockgröße benötigen, z.B. Verschlüsselungsalgorithmen.

[4] http://mmep.ite.fh-wiesbaden.de/info/rtp.htm, 28.04.09
[5] http://de.wikipedia.org/wiki/Real-time_Transport_Protocol, 01.04.09
[6] http://de.wikipedia.org/wiki/Real-time_Transport_Protocol, 01.04.09

- *Extension (X), 1 Bit*
  Das Erweiterungs-Bit ist gesetzt, wenn der Header um genau einen Erweiterungs-Header ergänzt wird.
- *CSRC Count (CC), 4 Bit*
  Der CSRC-Zähler gibt die Anzahl der CSRC-Identifier an.
- *Marker (M), 1 Bit*
  Das Marker-Bit ist für anwendungsspezifische Verwendungen reserviert.
- *Payload Type (PT), 7 Bit*
  Dieses Feld beschreibt das Format des zu transportierenden RTP-Inhalts (payload).
- *Sequence Number*
  Die Sequenznummer wird für jedes weitere RTP-Datenpaket erhöht. Die Startnummer wird zufällig ausgewählt und ist nicht vorherbestimmbar. Der Empfänger kann mit Hilfe der Sequenznummer die Paketreihenfolge wiederherstellen und den Verlust von Paketen erkennen.
- *Timestamp, 32 Bit*
  Der Zeitstempel gibt den Zeitpunkt des ersten Oktets des RTP-Datenpakets an. Der Zeitpunkt muss sich an einem Takt orientieren, der kontinuierlich und linear ist, damit die Synchronität des Streams sichergestellt und Laufzeitunterschiede der Übertragungsstrecke (Jitter) ermittelt werden können. Der Startwert sollte wie die Sequenznummer ein zufälliger Wert sein. Aufeinanderfolgende Pakete können den gleichen Zeitstempel haben, wenn die transportierten Daten z. B. zum selben Videoframe gehören. Pakete mit aufeinanderfolgenden Sequenznummern können aber auch nicht aufeinanderfolgende Zeitstempel enthalten, wenn wie z. B. bei komprimiertem Video Übertragungs- und Wiedergabereihenfolge nicht übereinstimmen.
- *SSRC, 32 Bit*
  Dieses Feld dient zur Identifikation der Synchronisationsquelle. Der Wert wird zufällig ermittelt, damit nicht zwei Quellen innerhalb der RTP-Session die gleiche Identifikationsnummer besitzen.
- *CSRC List, 0 bis 15 Felder je 32 Bit*
  Die CSRC-Liste dient zur Identifikation der Quellen, die im RTP-Payload enthalten sind. Die Anzahl der Listenfelder wird im CC-Feld angegeben. Falls mehr als 15 Quellen vorkommen, werden nur 15 identifiziert. Die Liste wird von Mixern eingefügt, die dazu den Inhalt des SSRC-Feldes der beteiligten Quellen einsetzen.

Wie der Header in der Protokollimplementierung zu bilden ist, wird vom RFC also genau vorgegeben, womit eine problemlose Modellierung möglich ist. Um eine korrekte Serialisierung des Protokollheaders in NS-3 und damit die Erkennung in PCAP Files zu ermöglichen, bietet NS-3 für diese Punkte eine vorgegebene Schnittstelle in Form der Klasse Header, von welcher Protokollheader eigener Protokolle ableiten müssen. Werden die verlangten Funktionen korrekt implementiert, ergeben sich dadurch folgende Vorteile:

- korrekte Serialisierung
  Der Header wird in PCAP- und Rohdaten-Traces korrekt gespeichert und kann z.B. in Wireshark als solcher eingesehen werden.
- Erkennen der eigenen RTP Implementierung als Protokoll in NS-3
  Werden eigene Protokollheader von Header abgeleitet, so weiss NS-3, um was für eine

Klasse es sich handelt und dass ein entsprechendes Protokoll vorhanden ist. Damit lassen sich auch Simulationsergebnisse aufgrund dieser Informationen analysieren (z.B. Filtern nach Protokoll).

2.2.3 RTP Socket

Im vorigen Kapitel wurde beschrieben, dass der Verkehrsgenerator modular programmiert werden soll und das verwendete Protokoll für den generierten Verkehr durch die Wahl eines entsprechenden Sockets bestimmt wird. Damit also der Traffic-Generator Voice-over-IP Verkehr in Form von RTP-Paketen generieren kann, muss ihm ein RTP Socket zur Verfügung gestellt werden, der den zu versendenden Paketen einen passenden RTP-Header beifügt. Dann werden die Pakete vom RTP-Socket über einen bereits bestehenden UDP-Socket verschickt. Die Implementierung dieses Sockets unter Einhaltung der von NS-3 vorgegebenen Socket-Schnittstelle ist einer der aufwändigsten Bestandteile des zu erstellenden Programmes.

2.2.4 Codecs

Bei der Implementierung von RTP gibt es nebst dem Header einen weiteren wichtigen Aspekt, der beachtet werden muss: Der RTP-Header gibt im Payload-Feld vor, welcher Typ von Anwenderdaten transportiert werden. Dieser Typ entspricht einem von RTP unterstützten Codec und setzt je nachdem gewisse Anforderungen an die Datenübertragung[7]. So verlangen einige Codecs z.B. feste Übertragungsraten und Payloadgrössen, was, wie in den vorherigen Kapiteln erklärt, dem Traffic-Generator über die Parametrisierungsschnittstelle mitgeteilt werden muss.

2.3 Zufallszahlen

Im vorigen Kapitel zum Traffic-Generator wurde die Parametrisierung durch Zufallszahlen und Verteilungen besprochen. Dieses Kapitel soll nun aufzeigen, welche Probleme sich aus der Verwendung von Zufallszahlen im Programm ergeben können und wie diese behoben werden können.

2.3.1 Generieren von Zufallszahlen

2.3.1.1 Echte Zufallszahlen

Echte Zufallszahlen zu generieren ist seit jeher ein Problem in der Informatik, und es existieren viele verschiedene Lösungen dafür. Die zuverlässigsten stützen sich dabei auf Hardware-Entropie-Quellen, die ihre Zufallswerte aufgrund des aktuellen Zustands der Geräte bestimmen[8], so z.B.

- Tastatureingaben und Mauszeigerbewegungen
- Gerätespannungen (Soundkarten, Laufwerke)
- Luftströmungen und -wirbel in Festplattenlaufwerken

Auch in NS-3 sind Zufallszahlen für diverse Netzwerksimulationen schon seit Projektbeginn eines der Hauptthemen, und das Programm verlässt sich in dieser Hinsicht auf die Funktionalität der /dev/random Schnittstelle, die Linux für die Generierung von Zufallszahlen aufgrund solcher Hardware-Entropie-Quellen zur Verfügung stellt. Dessen Funktionalität verpackt NS-3 in sogenannte Random-Klassen, die durch das ganze Programm hindurch immer wieder verwendet werden. Dabei verhalten sich diese Klassen wie Pseudozufallszahlen-Generatoren, die für ihre Seed-Werte Hardwarezufallszahlen aus der /dev/random-Schnittstelle verwenden.

---

[7] http://www.cisco.com/en/US/tech/tk652/tk698/technologies_tech_note09186a0080094ae2.shtml, 02.04.09
[8] Key Material and Random Numbers, Andreas Steffen, Seite 8, 19.02.2009

Im Rahmen dieser Bachelorarbeit genügt die von NS-3 zur Verfügung gestellte Funktionalität vollauf, denn sämtliche Simulationsaspekte in NS-3, wie Jitter, Paketausfälle, etc. basieren auf diesen Zufallszahlen. Ohne deren Implementierung zu prüfen, lässt sich also sagen, dass Programme, die dieses Modul verwenden, dem Niveau der gesamten Simulation entsprechen.

2.3.1.2 Zufallszahlen transformieren

In Linux ist /dev/random als Zufallszahlengenerator ohne besondere statistische Auffälligkeiten konzipiert, soll also einer uniformen Verteilung möglichst ähnlich sein. Obwohl nicht alle Entropiequellen, die der Generator dabei verwendet, diesen Anforderungen gerecht werden, erreicht er dieses Verhalten durch die Verwendung von Entropie-Pools und Hash-Algorithmen.[9]

Solche uniform verteilte Zufallszahlen sind allerdings nicht in jeder Situation geeignet. Auch für die im Kapitel zum Traffic-Generator beschriebene Verwendung von Zufallszahlen zur Bestimmung der Verkehrseigenschaften kann es erwünscht sein, dass eine Zufallsvariable z.B. häufiger grosse als kleine Pakete erzeugt. Das entspräche nämlich eher der Realität, da Netzwerkgeräte meist versuchen, den Overhead durch Paket-Header möglichst gering zu halten.

Um dieses Verhalten zu erzeugen, muss eine Zufallsvariable erstellt werden, die einer anderen Verteilungsfunktion folgt. Dazu gibt es zwei Lösungsansätze:

- Eine Entropiequelle finden, die der gewünschten Verteilung entspricht
- Die Uniform verteilte Zufallsvariable mithilfe mathematischer Transformationen zur gewünschten Verteilung umformen

Obwohl beider Verfahren durchaus denkbar sind, kann es sich als schwierig erweisen, eine natürliche Entropiequelle zu finden, die in allen Parametern exakt einer gewünschten mathematischen Funktion entspricht – ganz besonders, wenn diese Parameter flexibel anpassbar sein sollen.

Aus diesem Grund erzeugt NS-3 seine nicht-uniformen Zufallszahlen nach dem zweiten Verfahren und stellt mithilfe einiger Transformationsalgorithmen folgende Verteilungen zur Verfügung:

- Exponentialverteilung
- Paretoverteilung
- Normalverteilung
- Log-Normalverteilung
- Triangularverteilung

Die Zufallszahlen in NS-3 decken somit alle Anforderungen ab, die das Projekt diesbezüglich stellt.

2.3.1.3 Pseudozufallszahlen

Pseudozufallszahlen unterscheiden sich von echten Zufallszahlen dahingehend, dass nicht jeder Wert direkt aus einer zufälligen Entropiequelle stammt. Stattdessen wird aufgrund eines gegebenen oder zufälligen Startwerts (dem Seed) durch eine mathematische Funktion eine Reihe von scheinbar zufälligen Zahlen gebildet. Diese PRNGs (Pseudo-Random-Number-Generators) sind deterministisch und verletzen damit eigentlich das Prinzip der Zufälligkeit, da bei Kenntnis ihres internen Zustandes die nächsten generierten Zahlen vorhergesagt werden können. Nichtsdestotrotz werden PRNGs auch in Systemen eingestellt, wo echte Zufälligkeit kritisch ist (z.B. Verschlüsselungsalgorithmen), da sie

[9] Analysis of the Linux Random Number Generator, Gutterman & Pinkas, Kapitel 2 ff.

bei korrekter Implementierung und einem echt zufälligen Seed-Wert die folgenden Anforderungen an echte Zufallszahlen[10] erfüllen:

- Generieren von Zahlensequenzen mit einer sehr hohen Wahrscheinlichkeit, dass keine gleichen, nachfolgenden Werte auftreten.
- Generieren von Zahlensequenzen, die durch statistische Tests nicht von echt zufällig generierten Zahlenreihen unterscheidbar sind. Beispiele für statistische Tests sind:
  - Chi-Quadrat Test[11]
  - Monobit Test[12]
  - Runs Test[13]
- Unmöglichkeit, aufgrund einer Subsequenz der generierten Zahlen auf vorherige oder nachfolgende Zahlen oder sogar den internen Zustand des Generators zu schliessen.
- Unmöglichkeit, bei Kenntnis des momentanen internen Zustands des Generators auf vorherige Werte oder innere Zustände zu schliessen.

Alle Zufallszahlen in NS-3 sind als PRNGs realisiert und bieten zusätzlich die Möglichkeit, mit echt zufälligen Seeds aus der /dev/random-Quelle zu beginnen. Für Experimente und Simulationen legen PRNGs aber noch eine weitere, wichtige Eigenschaft an den Tag: Durch Wählen desselben Seeds können beliebig oft dieselben Reihen von Pseudozufallszahlen generiert werden. Somit lassen sich Experimente und Simulationen wiederholbar mit stets denselben, scheinbar zufälligen Zahlen durchführen.

2.3.2 Empirische Zufallszahlen

Ein letztes wichtiges Kapitel bei der Generierung von Zufallszahlen stellen empirische Zufallsvariablen dar. Empirisch bedeutet in diesem Fall, dass die Werte aufgrund von Erfahrungswerten und gemessenen Ergebnissen bestimmt werden. Konkret bedeutet das, dass man mit empirischen Zufallsvariablen ein gemessenes Verhalten in einem Netzwerk untersuchen kann und danach im Experiment immer wieder dieselben Lastverteilungen und -verhältnisse erzeugen kann, ohne dabei die exakt gleichen Werte zu erhalten.

Für TraGiC wäre diese Funktion beispielsweise interessant, um aus einer gegebenen Messung, wie etwa einer PCAP-Datei, eine Zufallsvariable zu erzeugen, die in der Simulation dieselben Lasteigenschaften erzeugt wie im gemessenen Original. Damit liessen sich bestehende Netzwerke sehr einfach modellieren und das Verhalten unterschiedlicher Komponenten unter realen Bedingungen beobachten. Obwohl diese Funktion nicht zwingend in der Aufgabenstellung verlangt wird, ist diese Möglichkeit, besonders im Hinblick auf zukünftige Projekte, interessant genug, um sich mit diesen empirischen Zufallsvariablen zu befassen.

Dabei zeigt sich erneut, dass die Zufallsvariablen in NS-3 bereits sehr ausgereift sind, denn eine empirische Zufallsvariable, die exakt die oben beschriebenen Eigenschaften hat, ist bereits vollständig in NS-3 implementiert.

---

[10] Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators, Bundesamt für Sicherheit in der Informationstechnik, http://www.bsi.de/zertifiz/zert/interpr/ais20e.pdf

[11] Der CHI-Quadrat Test, Blenkers & Becker, http://www.audicon.net/downloads/artikel/Chi-Quadrat-Test.pdf

[12] A statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST Special Publication 800-22, Andrew Rukhin et al., Kapitel 2.1, http://www.random.org/statistics/SP800-22b.pdf

[13] A statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST Special Publication 800-22, Andrew Rukhin et al., Kapitel 2.3, http://www.random.org/statistics/SP800-22b.pdf

2.3.2.1 Chi-Quadrat Test für empirische Zufallsvariablen in NS-3

Um die Syntax dieser empirischen Zufallsvariablen kennenzulernen und die Korrektheit ihrer Resultate zu überprüfen, wird ein Chi-Quadrat Unit-Test im TraGiC-Testprojekt eingebaut. Es ist die einzige Zufallsvariable, die so ausgiebig geprüft wird, da sie vom NS-3-Simulator nicht so häufig verwendet wird wie die übrigen Verteilungen und eventuelle Fehler noch nicht entdeckt sein könnten.

Der Chi-Quadrat Test basiert auf der gleichnamigen Chi-Quadrat-Verteilung. Das Prinzip dabei ist, eine durch eine Zufallsvariable generierte Menge von Zufallswerten zu prüfen und das Auftreten der einzelnen Werte zu zählen. Dieser gezählte Wert wird jeweils mit dem für die beschriebene Verteilung erwarteten Wert verglichen und die Abweichung davon registriert. Weichen die Werte dabei zu weit von einer durch die Chi-Quadrat-Verteilung vorgegebenen Grenze ab, folgt die Variable mit hoher Wahrscheinlichkeit nicht der verlangten Verteilung.

Der Chi-Quadrat Unit-Test geht hierbei wie folgt vor:

1. Erstellen einer empirischen NS-3-Zufallsvariable mit folgenden Wahrscheinlichkeiten:
   - $P(\{\omega: X(\omega) \leq 0\}) = 0$
   - $P(\{\omega: X(\omega) \leq 2\}) = 0.5$
   - $P(\{\omega: X(\omega) \leq 7\}) = 0.75$
   - $P(\{\omega: X(\omega) \leq 10\}) = 1$
   - Man beachte dabei:
     - Der Wertebereich dieser Variable liegt also zwischen 0 und 9 (Integer-Rundung)
     - Der exakte Wert 10 hat die Wahrscheinlichkeit 0, da einzelne Werte in einem Wahrscheinlichkeitsintegral keine Wertigkeit haben

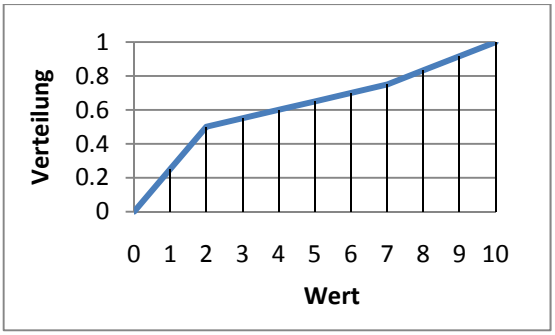


Abbildung 7: Verteilungsfunktion der Variable

2. Generieren von 100‘000 Zufallswerten mit dieser Variable. Dabei die Häufigkeit der Werte 0 bis 9 zählen.
3. Die erwarteten Vorkommen der Werte bestimmen:

Wert	Erwartete Anzahl
0	25‘000
1	25‘000
2	5‘000
3	5‘000
4	5‘000
5	5‘000

6	5‘000
7	8‘333
8	8‘333
9	8‘334

Tabelle 4: Verteilungsmuster von 100'000 Zufallswerten

4. Die Chi-Quadrat Prüfsumme berechnen:
   - $\chi^2 = \sum_{i=1}^{n} \frac{(h_i - h_E)^2}{h_E}$
     - $h_i =$ effektive Anzahl Wert i
     - $h_E =$ erwartete Anzahl Wert i
5. Das 0.95-Quantil der χ^2-Verteilung mit 9 Freiheitsgraden ($k = n - 1$) bestimmen ($= 16.92$).
6. Folgt die Zufallsvariable der erwarteten Verteilung, darf die Chi-Quadrat Prüfsumme das vorgegebene Quantil nicht überschreiten (95% Zuverlässigkeit)
   - Im Testfall beträgt die Prüfsumme 6.13 und unterschreitet den vorgegebenen Wert somit deutlich.

3 Architekturdesign und Implementierung

3.1 Anforderungen

Für das zu erstellende Grunddesign gehen aus dem Kapitel „Problemstellung und Ziele“ verschiedene, für die Architektur relevante Anforderungen im Bereich Erweiterbarkeit, Modifizierbarkeit, Funktionalität und Integration in NS-3 hervor. Diese Anforderungen seien in den folgenden Unterkapiteln kurz hervorgehoben:

3.1.1 Integration in NS-3

TraGiC ist als Erweiterung zum NS-3 Netzwerksimulationssystem konzipiert und muss eng mit dessen Klassen und Methoden zusammenarbeiten, um die gegebenen Anforderungen zu erfüllen. Aus diesem Grund müssen beim Design von vornherein spezifische, von NS-3 vorgegebene Architekturrichtlinien beachtet werden. Dazu gehören:

- Respektieren der Klassenhierarchie
  - Applikationen wie der TrafficGenerator müssen stets von ns3::Application ableiten, um in der von NS-3 bereitgestellten Laufzeitumgebung fungieren zu können.
  - Protokollheader müssen von ns3::Header ableiten, um eine korrekte Serialisierung und Darstellung im Programm zu gewährleisten.
- Verwenden von Applikationshelpern
  - Eigene Applikationen müssen parallel einen Helper mitbringen, welcher die Installation der selbigen auf Nodes und NodeContainern erleichtert.

3.1.2 Erweiterbarkeit

Der geplante Verkehrsgenerator bietet in seiner Grundfunktion lediglich die Möglichkeit, einzelne Pakete unterschiedlicher Grössen an verschiedene Ziele im Netzwerk zu versenden. Während dieses simple Verhalten allein in praktischen Simulationen nur selten Verwendung findet, bietet es dennoch die Grundlage zur einfachen Spezifizierung komplexerer Applikationen und Abläufe. Architektonisch muss TraGiC folglich Spielraum für folgende Erweiterungen lassen:

3.1.2.1 Austauschen des verwendeten Protokolls

Welches PDUs zum Versenden der generierten SDUs benutzt werden, muss gänzlich unabhängig vom eigentlichen TrafficGenerator spezifiziert werden können. Es soll den TrafficGenerator weder beeinflussen noch muss er sich in spezieller Weise um die Auswahl oder Implementierung eines Protokolls kümmern.

3.1.2.2 Implementierung spezifischer Applikationen

Der TrafficGenerator muss eine Schnittstelle anbieten, über die ihm mitgeteilt werden kann, wann er welches Paket an welches Ziel senden soll. Die Koordination der einzelnen Verkehrsgeneratoren untereinander, welche letztlich das gewünschte Verkehrsmuster erzeugt, bleibt dabei dem Benutzer überlassen.

3.1.2.3 Komplexe Simulationen

Verwendete Verkehrsgeneratoren müssen auch in komplexeren Netzwerken beliebig kombinier- und verwendbar bleiben. Die Anzahl Verkehrsgeneratoren darf nicht von der Architektur vorgeschrieben respektive eingeschränkt werden.

3.1.3 Modifizierbarkeit

Für spezifische Erweiterungen des Generators, wie sie im vorherigen Kapitel beschrieben wurden, gilt die Regel, dass auch spezialisierte Verhaltensweisen konfigurierbar bleiben müssen. Als Beispiel sei hierbei die von TraGiC bereitgestellte VoipTrafficGenerator-Implementierung erwähnt. Bei diesem Szenario sind VoIP-spezifische Verhaltensmuster, wie etwa der automatische Aufbau einer Duplex-Verbindung zwischen 2 Verkehrsgeneratoren (Gespräch) fest vorgegeben. Dennoch behält der Benutzer die Möglichkeit, Rahmenparameter, wie etwa die mittlere Gesprächsdauer, den durchschnittlichen Abstand zwischen zwei Gesprächen oder das zu verwendende Medien-Codec zu verändern und an das gewünschte Verhalten anzupassen.

3.2 Design

Bei neuen Protokollimplementierungen gibt NS-3 bereits eine Struktur vor, von der TraGiC nicht abweichen soll. Auch dass eine Klasse TrafficGenerator als Application implementiert werden muss, ist durch die Anforderungen und vorangegangenen Überlegungen klar, so dass sich folgendes grundlegendes Design ergibt, welches bei allen Designvarianten gleich modelliert sein muss:

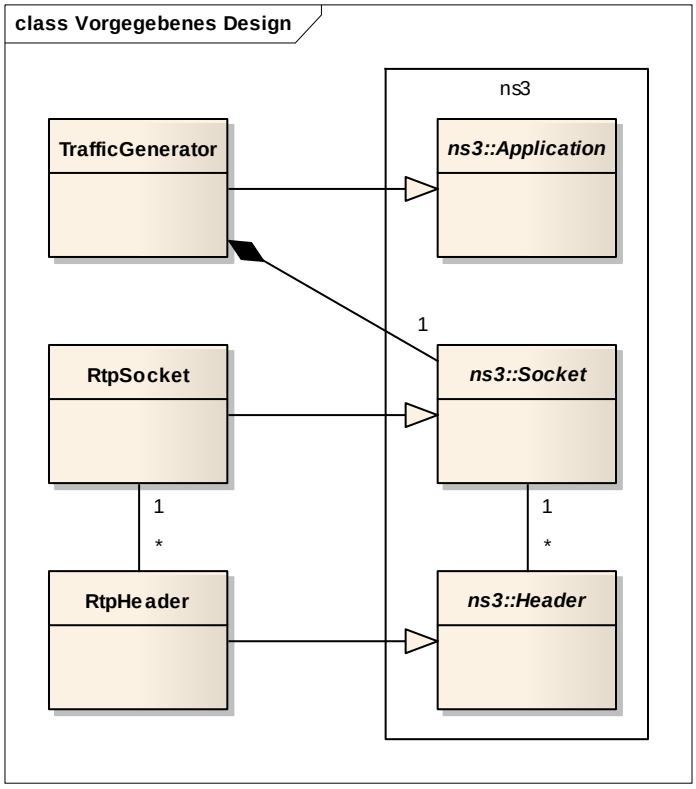


Abbildung 8: Vorgegebenes Design von NS-3

3.2.1 Designvariante 1

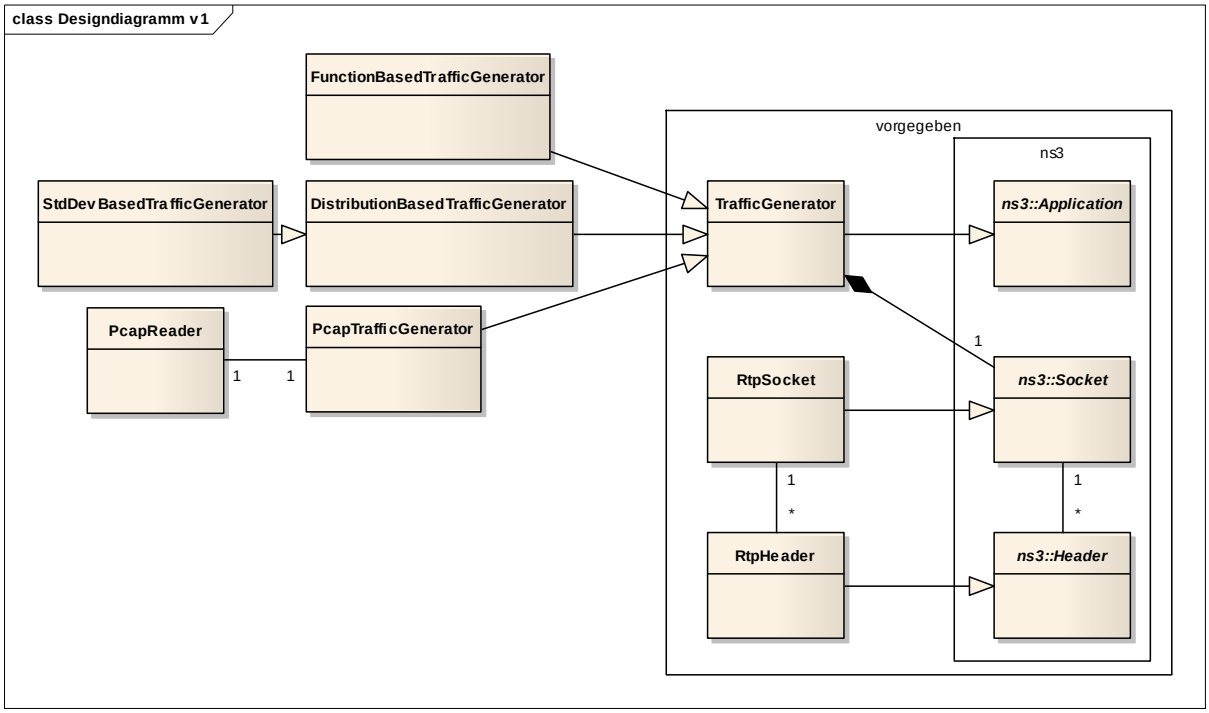


Abbildung 9: Designvariante 1

Hier werden verschiedene Varianten der Verkehrsgenerierung (andere Verteilungsfunktionen) direkt als eigene Generatoren von TrafficGenerator abgeleitet. Dabei entstehen verschiedene Unterklassen, welche beispielsweise auf einer Standardnormalverteilung basieren könnten oder Ergebnisse der Analyse eines PCAP Files als Verteilung aufgreifen (empirische Verteilung). Es könnte eine Funktion für die Berechnung der Parameter angegeben werden (FunctionBasedTrafficGenerator), oder der Generator basiert auf Verteilungen (DistributionBasedTrafficGenerator und die Variante StdDevBasedTrafficGenerator, welcher die Standardnormalverteilung einsetzt). Denkbar wäre auch ein Generator, der auf empirischen Zahlen – einem PCAP File – basiert (PcapTrafficGenerator, der die Klasse PcapReader zum Einlesen von PCAP Files verwendet). VoIP-spezifische Aufgaben wie Call-Management übernimmt die TrafficGenerator-Klasse.

3.2.2 Designvariante 2

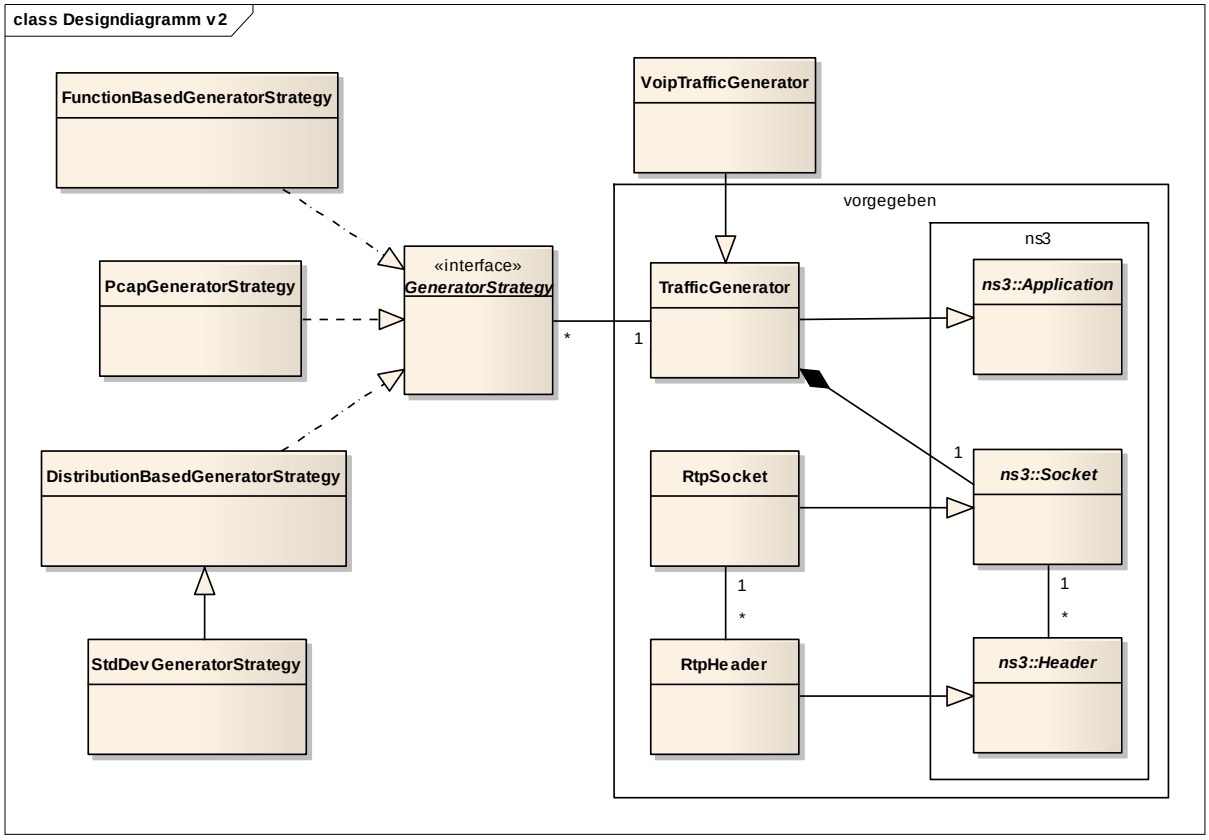


Abbildung 10: Designvariante 2

Bei diesem Design wird das Strategy Pattern[14] angewendet. Spezifische Generatoren werden von TrafficGenerator abgeleitet und beinhalten für jeden einstellbaren Parameter eine Strategie beziehungsweise einen Algorithmus, um ihn für jedes Paket zu berechnen. Ein konkreter Generator (hier: VoipTrafficGenerator) kann somit für die Paketgrösse eine FunctionBasedGeneratorStrategy verwenden, um völlig zufällige Payloads zu erhalten, für den Zeitabstand zwischen den Paketen aber eine Verteilungsfunktion wählen (DistributionBasedGeneratorStrategy und StdDevGeneratorStrategy für die Standardnormalverteilung). Mit der PcapGeneratorStrategy können Erfahrungswerte eingefügt werden. VoIP-spezifische Aufgaben wie Call-Management übernimmt die VoipTrafficGenerator-Klasse, womit die Klasse TrafficGenerator generisch bleibt.

[14] Über eine Schnittstelle werden mehrere austauschbare Algorithmen definiert, http://de.wikipedia.org/wiki/Strategy_pattern, E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, 1994

3.2.3 Designvariante 3

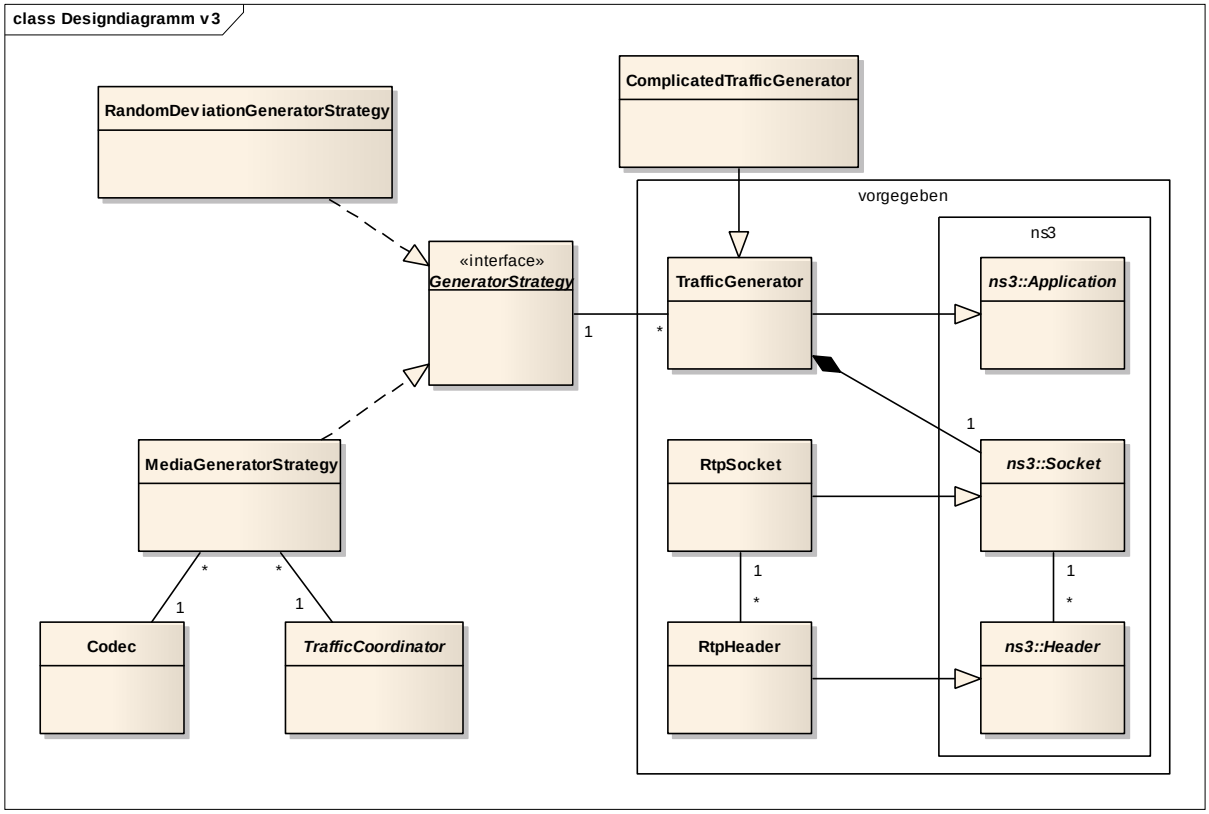


Abbildung 11: Designvariante 3

Wie bei Variante 2 findet auch hier das Strategy Pattern Verwendung. Eine Strategie ist hier „intelligent“, da sie sämtliche abfragbaren Parameter kapselt. Die Realisierungen des GeneratorStrategy-Interfaces kümmern sich um die protokollspezifischen Aufgaben. Im Falle von VoIP wird das Call-Management über einen globalen VoipTrafficCoordinator koordiniert. Ein spezialisierter TrafficGenerator wird damit bei RTP nicht benötigt, da sämtliche Parameter von der eingesetzten Strategie abhängen. Eine solche Strategie kann definiert werden und mehreren Generatoren aggregiert werden, um mehrere gleich agierende Nodes zu schaffen.

3.2.3.1 Codec

Für die Implementierung der verschiedenen in TraGiC gebräuchlichen RTP-Codecs kommt eine Klasse Codec in Frage, die losgelöst von der ganzen Architektur ist. Darin werden lediglich die codecspezifischen Werte (Payload Type, Paketrate, Paketgrösse) festgelegt. Abgefragt werden diese Werte von der Strategie. Der Benutzer entscheidet beim Erstellen der Strategie, welchen Codec er verwenden möchte.

3.2.3.2 Media / VoIP

Durch die Trennung der VoIP Funktionalität von der einfachen RTP Funktionalität lässt sich die Strategie generell implementieren und das effektive Verhalten vom Coordinator bestimmen. Eine MediaGeneratorStrategy beherrscht somit den Umgang mit einem Codec, wie bei RTP üblich, aber die Streamziele werden von einem bestimmten Coordinator vorgegeben. Mit dem Einsatz eines VoipCoordinators an dieser Stelle können VoIP-spezifische Regelungen wie besagte Duplexverbindungen implementiert werden. Ein MediaCoordinator verschickt einfache RTP-Streams, ohne sich um VoIP zu kümmern, wie es beispielsweise bei einem Videostream der Fall ist.

3.2.3.3 TrafficCoordinator

Um die in der Analyse beschriebene Verkehrsverteilungstabelle in TraGiC zu implementieren, bietet sich eine globale Klasse TrafficCoordinator an, welche der MediaGeneratorStrategy eines TrafficGenerators als Pointer übergeben wird. Somit können sämtliche Strategien auf diese Tabelle zugreifen und die benötigten Parameter auslesen. Eine Alternative ist, die Klasse als static zu markieren. Dies bietet den Vorteil, dass die Tabelle applikationsweit gilt und die Konfiguration und Programmierung vereinfacht wird. Im Hinblick auf die Erweiterungsmöglichkeit von TraGiC wurde jedoch für die erste Variante entschieden. Auch wenn die Verteilungstabelle global definiert ist und bei allen Nodes dieselbe sein muss, kann man auf diese Weise jederzeit die Zielverteilung ändern oder, falls die Verteilungstabelle nicht mehr erwünscht ist, ein gänzlich anderes Konzept verwenden, da der Code entkoppelt ist.

3.2.3.3.1 VoipTrafficCoordinator

Spezifisch für VoIP wird auch ein Call-Management benötigt: Zwei Nodes, die miteinander kommunizieren, haben stets in beide Richtungen Verkehr. Dazu muss sichergestellt werden, dass Nodes jeweils nur ein Gespräch gleichzeitig führen. Diese Aufgaben übernimmt ein spezieller VoipTrafficCoordinator, indem er sämtliche Calls (bidirektionale Streams) mit Anfangs- und Schlusszeit in einer Liste speichert und bei Call-Anfragen nachschaut, ob überhaupt ein freier Node verfügbar ist. Ist dies nicht der Fall, so wird dem anfragenden Node die Loopback-Adresse 127.0.0.1 übergeben, was bewirkt, dass der Node für die Dauer seines Calls inaktiv wird.

Ein weiteres Problem ergibt sich beim Scheduling der Calls. Als Beispiel wird ein Node N von den Nodes A, B und C angerufen, jeweils zu unterschiedlichen Zeitpunkten. Folgende Abbildung verdeutlicht die Situation.

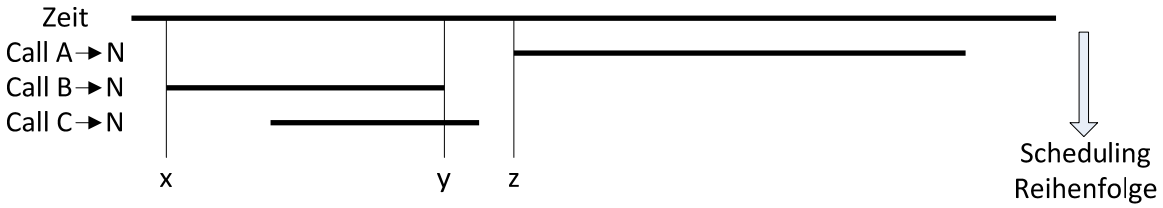


Abbildung 12: Call Scheduling

Das Scheduling erfolgt in der Reihenfolge A, B, C, so dass zuerst der Anruf von A abgehandelt wird und durch die Duplexverbindung beide Nodes als besetzt gelten. Der Anruf von A ist jedoch erst zu einer späteren Zeit z geplant, so dass es sein kann, dass zuvor noch andere Anrufe auf N eingehen könnten, sofern sie vor dem bereits geplanten Anruf beendet werden. Dies wird durch den Anruf von B an N repräsentiert, der zu einem Zeitpunkt x beginnt und bei y beendet ist.
Nun muss N „geweckt“ werden, da dieser Generator erst zum Zeitpunkt z seine Aktivität wieder aufnehmen würde. Dies bewirkt, dass der Generator erneut bei der Strategie (und damit dem Coordinator) seine nächste Zieladresse sowie die Zeit bis zum Senden abfragt.
Da nun beide Calls in die Liste eingetragen und als gut befunden wurden, kann C den Node N nicht mehr anrufen, da sein Call zwar vor A stattfindet, aber während dem Call von B beginnen würde.

3.2.4 Vergleich und Entscheid

Bei Designvariante 1 tritt ein Problem auf, wenn ein spezifischer Traffic Generator benötigt wird, beispielsweise für ein anderes Protokoll, bei dem jedes Paket bestätigt werden muss. Hier müssten Implementierungsdetails des Protokolls im Traffic Generator angesiedelt werden. Dieser sollte aber wiederum nicht wissen, welches Protokoll er bedient, da er generisch bleiben soll. Auch stellt sich die Frage, wo eine solche Klasse Platz finden würde, denn weder direktes Ableiten aus TrafficGenerator noch aus den Unterklassen mit den verschiedenen Strategien wäre eine saubere Lösung.
Desweiteren sollte ein Generator auch mehrere verschiedene Verteilungen für verschiedene Grössen unterstützen: Eine Standardnormalverteilung für die Paketgrösse und eine empirische Verteilung für die Zeitabstände wären denkbar.

Eine logische Folgerung ist Designvariante 2 mit dem Strategy Pattern, um für verschiedene Variablen auch verschiedene Verteilungen zur Verfügung zu stellen. Das Interface GeneratorStrategy definiert verschiedene Methoden für Zeitabstand zwischen Paketen, Zielbestimmung und andere Parameter, die bei jedem Paket wechseln können. Diese Lösung bietet sowohl für Programmierer als auch für Anwender den besten Komfort, da auf einfache Weise neue Strategien und neue Generatoren hinzugefügt werden können.
Ein Mangel ist hier aber die fehlende Unterteilung zwischen der Strategie und dem Generator, denn es ist nicht klar festgelegt, wo der protokollspezifische Code eingebaut werden soll. Auch kann es sehr aufwendig werden, für jeden einzelnen Parameter eines Protokolls eine eigene Strategie zu implementieren, wodurch der gewonnene Komfort schnell zur Qual werden kann.

Designvariante 3 löst diese Probleme, indem sämtliche protokollspezifischen Parameter in einer einzelnen Strategie gekapselt werden. Damit hat man an einem einzigen Ort den Überblick über die komplette Verhaltensweise des Traffic Generators, und bei einfacheren Protokollen wie RTP wird kein spezieller TrafficGenerator benötigt. Sollte es dennoch vonnöten sein, ein spezielles Verhalten zu definieren, welches nicht über die Strategie gelöst werden kann, bietet sich immer noch eine Ableitung der generischen Klasse an.
Diese Variante bietet hohe Konfigurierbarkeit und gleichzeitig den Komfort für den Programmierer, bei einer neuen Implementierung lediglich eine neue Strategie zu definieren. Aus diesen Gründen wurde entschieden, Designvariante 3 für TraGiC zu wählen.

3.3 Sequenzdiagramme

Aufgrund des endgültigen Designs ergibt sich ein sehr geradliniger Programmablauf für die beiden zentralen Komponenten von TraGiC, dem Traffic-Generator und dem RTP-Header. Diese seien im Folgenden kurz durch ein Sequenzdiagramm beschrieben:

3.3.1 Traffic Generator

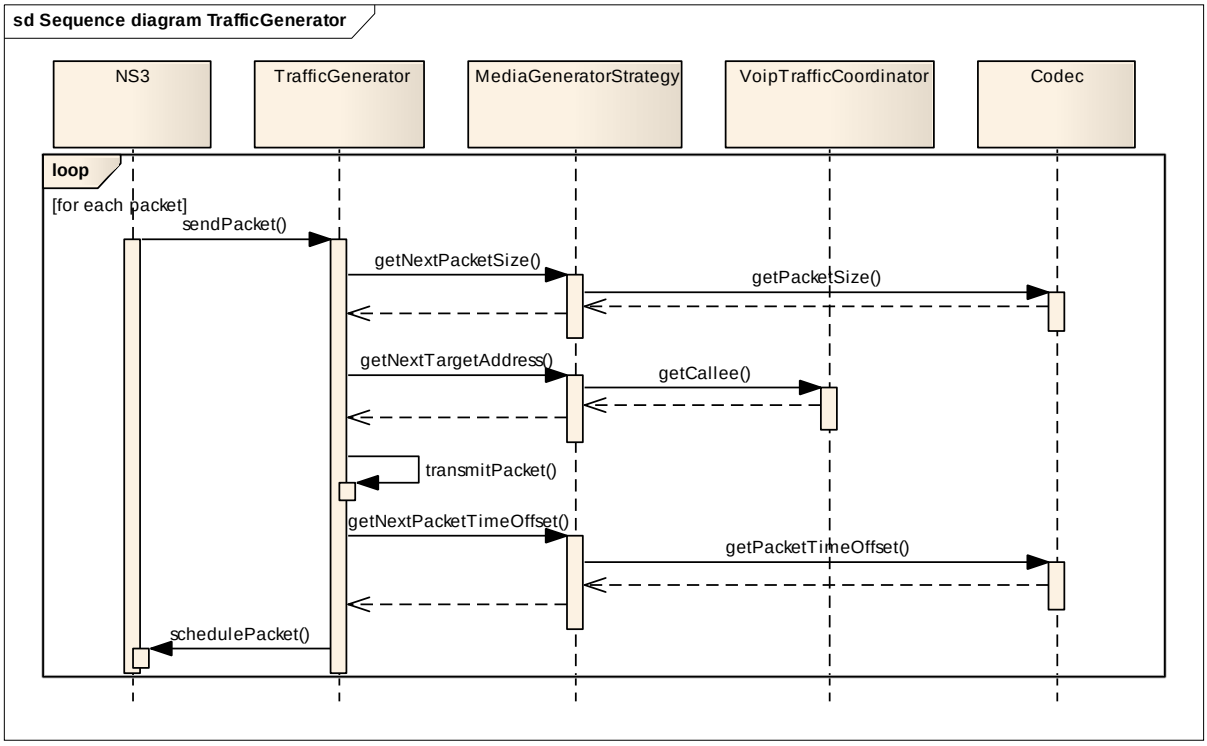


Abbildung 13: Sequenzdiagramm für den Traffic Generator

Beim Versenden von Paketen hält sich der TrafficGenerator an die ihm von seiner Strategy zugewiesenen Parameter. Diese beinhalten den Zeitabstand bis zum Versenden des nächsten Pakets, die Grösse desselben und die Zieladresse für die Übertragung. Wie die Strategy zu diesen Werten kommt, bleibt ihr überlassen hängt stark von der gewählten Implementierung ab. So existieren z.B. Strategien, die alle Werte völlig zufällig wählen, wohingegen die MediaGeneratorStrategy in der obigen Grafik diese Werte aufgrund von Codec-Einstellungen und den Angaben ihres Koordinators ableitet. Hat der TrafficGenerator alle benötigten Daten erhalten, erstellt er das zu versendende Paket und meldet es beim Scheduler zum entsprechenden Zeitpunkt an. Anschliessend initiiert der Scheduler zum gegebenen Zeitpunkt das Versenden des soeben geplanten Pakets, was im folgenden Kapitel genauer illustriert wird.

Die Methode transmitPacket() wird im nachstehenden Diagramm genauer erläutert; es handelt sich um den Teil der Applikation, in dem der entsprechende Socket angesprochen wird und die RTP-Implementierung benötigt wird.

3.3.2 RTP

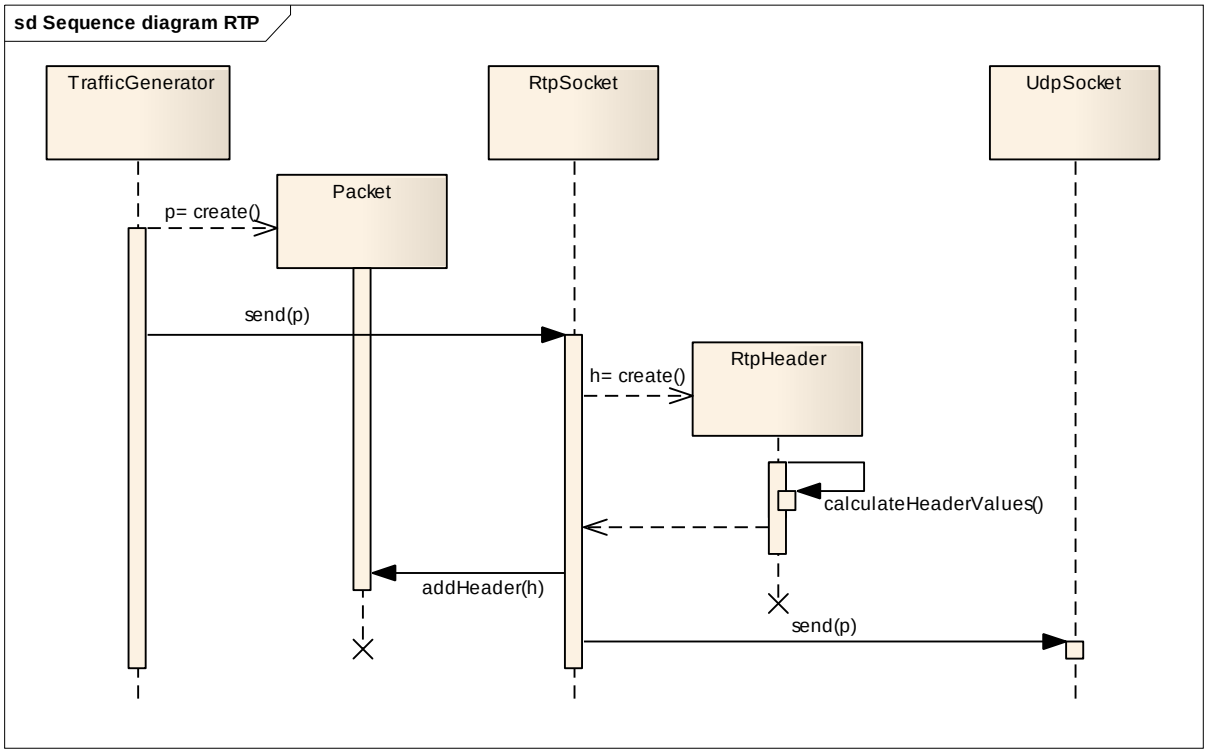


Abbildung 14: Sequenzdiagramm für das RTP Protokoll

Das Diagramm stellt dar, was innerhalb der transmitPacket()-Methode geschieht. Der TrafficGenerator kreiert das nächste zu versendende Paket und verschickt es über seinen RtpSocket via die Methode send(). In einem nächsten Schritt erzeugt der RtpSocket nun den für die Übertragung notwendigen, korrekten RtpHeader und fügt diesen zum Paket hinzu. Damit ist das RTP-Paket vollständig beschrieben und kann über einen gewöhnlichen UDP-Socket in die NS-3 Simulationsumgebung eingespeist werden.

4 Simulationsläufe

Zur Validierung des Systems werden im Folgenden einige Null- und Testläufe durchgeführt, welche eventuelle Schwächen in der Simulation aufzeigen sollen. Diese Simulationsläufe werden im Folgenden kurz erklärt und auf ihre Plausibilität überprüft.

In den untenstehenden Abbildungen sind Netzwerke durch Kreise und Verkehr dazwischen durch Pfeile dargestellt. Bitte beachten Sie dabei, dass die Abbildungen nicht massstabsgetreu sind und Grössenunterschiede bei den Netzwerken nicht auf unterschiedliche Nodezahlen schliessen lassen.

4.1 Null-Lauf: Gespräch zwischen 2 Nodes

4.1.1 Testaufbau

- 2 Netzwerke
  - A 1 Node
  - B 1 Node
- Point-to-Point Netzwerk
- Ø Gesprächsdauer: 300s
- Ø Gesprächsabstand: 10s
- Simulationsdauer: 300s
- Verkehrsverteilung:

	A	B
A	0	1
B	1	0

Tabelle 5: Verkehrsverteilung 2-Node-Gespräch

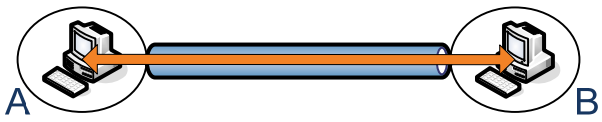


Abbildung 15: Gespräch zwischen 2 Nodes

4.1.2 Resultat

- Gemessene Erlangwerte:

	A	B
A	0	0.960333
B	0	0

Tabelle 6: Erlangwerte 2-Node-Gespräch

- Interpretation
  - Die beiden Nodes haben während der Simulation immer nur einen Gesprächspartner für ihren VoIP-Verkehrsgenerator. Es werden 2 Gespräche geführt, unterbrochen durch eine kurze, im Durchschnitt 10 Sekunden lange Pause. Aufgrund dieser Pause sind es auch nur 0.96 und nicht ganz 1 Erlang. In beiden Fällen ist A der Initiator des Gesprächs, da A als erstes den TrafficCoordinator um ein neues Ziel bittet.

4.2 Gleichverteilter Verkehr zwischen 3 Gruppen

4.2.1 Testaufbau

- 3 Netzwerke
  - A 5 Nodes
  - B 5 Nodes
  - C 5 Nodes
- CSMA-Netzwerk
- Ø Gesprächsdauer: 120s
- Ø Gesprächsabstand: 10s
- Simulationsdauer: 300s
- Verkehrsverteilung:

	A	B	C
A	1	1	1
B	1	1	1
C	1	1	1

Tabelle 7: Verkehrsverteilung gleichverteilter Verkehr

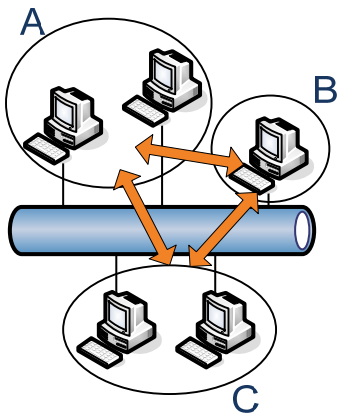


Abbildung 16: Gleichverteilter Verkehr zwischen 3 Gruppen

4.2.2 Resultat

- Gemessene Erlangwerte:

	A	B	C
A	0.991195	0.241233	0
B	0.0195594	0.930137	0.459646
C	0.162995	0.599821	0.559969

Tabelle 8: Erlangwerte gleichverteilter Verkehr

- Interpretation
  - Die Summe über alle Erlang-Werte beträgt 3.8178599, was etwa der Hälfte der theoretisch möglichen 7.5 Erlang entspricht. Diese Differenz entsteht zum einen durch die 10 Sekunden langen Pausen, die durchschnittlich alle 120 Sekunden am Ende eines jeden Gesprächs eingefügt werden. Zum anderen ist der Erlang-Verlust darauf zurückzuführen, dass nicht für jeden Node zu jedem Zeitpunkt ein gewünschter Gesprächspartner zur Verfügung steht. Obwohl jedes Zielnetzwerk für jeden Node gleich wahrscheinlich ist, kann nicht ausgeschlossen werden, dass zu

einem gegebenen Zeitpunkt in der Simulation mehr als 5 Nodes dazu entschliessen, in ein einzelnes Netzwerk zu senden. Ist dies der Fall, werden überzählige Nodes abgewiesen und es wird kein Verkehr generiert. Diese abgewiesenen Calls führen ebenfalls dazu, dass die gemessenen Erlangwerte nur ansatzweise mit den geplanten Verteilungswerten korrelieren.

4.2.3 Reaktion

Um den Effekt der abgewiesenen Calls zu mindern, wird dieselbe Simulation mit jeweils 50 Nodes pro Netzwerk durchgeführt. Wenn die vorherige Annahme richtig war, sollte dies dafür sorgen, dass jeweils mehr Nodes für Gespräche zur Verfügung stehen und temporäre Überbelastungen einzelner Netzwerke nicht zu sehr ins Gewicht fallen. Der gleiche Effekt liesse sich übrigens auch erzielen, indem man die Simulationszeit erhöhen würde.

- Gemessene Erlangwerte:

	A	B	C
A	6.00138	5.13906	5.19787
B	8.79521	6.89235	6.49341
C	8.48329	5.49929	6.37237

Tabelle 9: Erlangwerte gleichverteilter Verkehr (50 Nodes)

- Interpretation
  - Die neu gemessenen Erlangwerte zeigen eine wesentlich gleichmässigere Verteilung, wie sie aus der angegebenen Verteilungstabelle auch zu erwarten wäre. Ausserdem werden mit über 60 von 75 möglichen Erlang sehr viel weniger Calls verworfen und die Nodes somit besser ausgelastet. Berechnet man die durchschnittlichen Gesprächsabstände von etwa 10 Sekunden mit ein, so ergibt sich ein theoretisches Maximum von 68 Erlang, was durch die erreichten 60 Erlang ausserordentlich gut erreicht wird.

4.3 Überbeanspruchtes Netzwerk

4.3.1 Testaufbau

- 3 Netzwerke
  - A 20 Nodes
  - B 1 Nodes
  - C 9 Nodes
- CSMA-Netzwerk
- Ø Gesprächsdauer: 120s
- Ø Gesprächsabstand: 30s
- Simulationsdauer: 300s

- Verkehrsverteilung:

	A	B	C
A	3.6	6.4	10
B	0.5	0	0.5
C	3	3	3

Tabelle 10: Verkehrsverteilung überbeanspruchtes Netzwerk

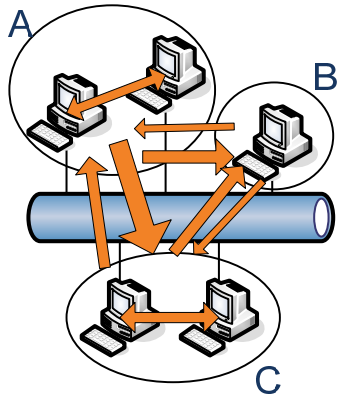


Abbildung 17: Überbeanspruchtes Netzwerk

4.3.2 Resultat

- Gemessene Erlangwerte:

	A	B	C
A	1.53555	0.0979165	4.82429
B	0	0	0
C	1.43393	0.781631	0.165887

Tabelle 11: Erlangwerte überbeanspruchtes Netzwerk

- Interpretation
    - Diese Messresultate zeigen deutlich auf, dass Netzwerk B völlig überlastet ist. Nicht ein einziger Call kann aus Netzwerk B in ein anderes Netzwerk geführt werden, denn der einzige Node, der darin verfügbar ist, wird ständig durch Anrufe aus anderen Netzwerken belegt. Diese Überbelegung des Netzwerks führt dazu, dass Netzwerk A seine geplante Last in Netzwerk B fast gar nicht und Netzwerk C sie nur bedingt erfüllen kann. Mit einer Summe von nur 8.8392045 von 15 möglichen Erlang zeigt sich ebenfalls, dass sehr viele Calls verworfen werden müssen, da der eine, benötigte Node aus Netzwerk B nicht zur Verfügung steht.

5 Zusammenfassung und Ausblick

5.1 Ergebnisse

5.1.1 Was wurde gemacht

Zusammenfassend setzt sich TraGiC in seiner jetzigen Version aus folgenden Komponenten zusammen.

- RTP
  - Protokollheader
  - Socket zur Mediendatenübermittlung
- Verkehrsgenerator
  - Basissystem
    - frei implementierbare GeneratorStrategy-Basisklasse
  - Medienstream-Verkehrsgenerator
    - über mittlere Streamdauer, Streamabstand und Verteilungsmatrix konfigurierbar
  - VoIP-Verkehrsgenerator
    - über mittlere Gesprächsdauer, Gesprächsabstand und Verteilungsmatrix konfigurierbar
- Graphische Visualisierung der Simulationsdaten
  - INAV[15]
  - SMART[16]
  - Wireshark[17]

Alle diese Komponenten wurden erfolgreich implementiert und getestet. Sie erfüllen die an das Projekt gestellten Anforderungen vollständig und übertreffen sie, besonders im Bereich der Visualisierung, sogar noch. Wir erachten das Projekt TraGiC somit als erfolgreich beendet und überlassen es den Projektauftraggebern, das Programm weiter zu prüfen und in den NS-3 Stammcode zurückzuführen.

5.1.2 Fehlende Module

Keine. Alle verlangten und geplanten Komponenten von TraGiC wurden implementiert und getestet.

5.2 Ausblick

5.2.1 Mögliche Erweiterungen

5.2.1.1 Parametereingabe als PCAP

Als mögliche Erweiterung für TraGiC wurde erwähnt, dass das Programm anstelle der vorgegebenen Mittelwerte und Verteilungen der Gespräche diese Daten auch aus echten Verkehrsmitschnitten, explizit PCAP-Files, entnehmen könnte. Nötig dafür wären in erster Linie ein PCAP-Reader, welcher in

[15] INAV – Interactive Network Active-traffic Visualization, http://inav.scaparra.com/, siehe Kapitel 6.1 „User Manual“
[16] SMART – Safe Mapping and Reporting Tool, http://safemap.sourceforge.net/, siehe Kapitel 6.1 „User Manual“
[17] Wireshark Network Protocol Analyzer, http://www.wireshark.org/

dieser Form noch nicht in NS-3 vorhanden ist, und statistische Methoden zur Berechnung der benötigten Verteilungs- und Mittelwerte aus den eingelesenen Daten.

Konkret geplant ist dieser Zusatz allerdings noch nicht, da die praktische Nützlichkeit so eines Systems noch mit den betreffenden Auftraggebern besprochen wird.

6 Verzeichnisse

6.1 Glossar

Application	Um Pakete über das in ns-3 modellierte Netzwerk zu versenden, müssen so genannte Applications Pakete generieren und über Netzwerksockets versenden. Der TraGiC-Verkehrsgenerator ist eine solche Applikation.
Channel	Channels in ns-3 stellen das Netzwerkmedium dar. So existieren beispielsweise CSMA-Channels oder WiFi-Channels. Channels haben in den meisten Fällen spezifische Eigenschaften, wie etwa Kollisionsgefahren oder Interferenzen.
cygwin	Cygwin ist ein GNU basiertes Buildsystem mit Microsoft Windows als Zielsystem, wobei cygwin versucht, eine Linux-typische Umgebung (z.B. in Bezug auf Linux-spezifische Header-Dateien) zu simulieren. Das ermöglicht es in bestimmen Fällen, für Linux geschriebenen Quellcode für Windows-Betriebssystem zu kompilieren. http://www.cygwin.com/
Device	Als Device wird ein Netzwerkgeräte-Objekt bezeichnet, das auf Nodes installiert wird und im Zugriff auf ein bestimmtes Medium ermöglicht. Es ist eine direkte Abstraktion einer Netzwerkschnittstelle.
GNU	**G**NU is **n**ot **U**nix Das GNU-Projekt wurde ins Leben gerufen, um ein vollständig freies Betriebssystem zu kreieren. Linux geht auf dieses Projekt zurück und es stellt die Grundlage für das GNU Buildsystem, das dazu entwickelt wurde, Programme für durch GNU entwickelte Betriebssysteme zu erstellen. http://www.gnu.org/
GNU Buildsystem	Sammlung von Tools für das Erstellen von Computerprogrammen. Ursprünglich nur für die Betriebssysteme des GNU-Projekts bestimmt, halten sich nun auch Buildsysteme für kommerzielle Betriebssysteme (z.B. MinGW für Windows) an den GNU-Standard. Wichtige Bestandteile des GNU Buildsystems sind: • gcc, C-Compiler • g++, C++ Compiler • make, abhängigkeitengesteuertes Buildsystem http://www.gnu.org/software/hello/manual/automake/GNU-Build-System.html
MinGW	MinGW ist eine Implementierung des GNU Buildsystem Standards zur Erstellung von Programmen für Windows. MinGW stellt einen wichtigen Teil der Bestrebungen der C++ Community dar, C++ Programme für alle Plattformen gleichermassen lauffähig zu erstellen, indem die verwendeten Makefiles für jedes Zielsystem verwendet werden können. http://www.mingw.org/
ns-3	Network Simulator 3 ns-3 ist ein auf diskreten Events basierender Simulator für Netzwerksysteme und wurde primär für Forschungs- und Bildungsbereich entwickelt. http://www.nsnam.org/
Node	Nodes bezeichnen in ns-3 ganz allgemein Netzwerkknoten. Dies können beliebige, aktive Netzwerkkomponenten, wie Router,

	Netzwerk-Clients oder sogar mobile Geräte darstellen. Nodes verfügen über eine variable Anzahl Devices, über die sie auf das Netzwerk zugreifen können.
Socket	Schnittstelle für ns-3 Applikationen zum Versenden von Paketen. Für jedes verfügbare Protokolle wird in der Regel ein eigener Sockettyp verwendet, der das verpacken von SDUs zu PDUs übernimmt.
Strategy	Strategy ist Software-Pattern der Gang of Four (GoF), das in TraGiC häufig im Zusammenhang mit der Implementierung der Verkehrsgenerator-Implementierung Anwendung findet. Es dient dazu, Verhaltensweisen von Modulen und Klassen austauschbar zu realisieren.
waf	Waf ist ein Python-basiertes C++ Buildsystem. Der grosse Vorteil von waf ist seine grosse Interoperabilität mit verschiedenen Betriebssystemen, da es lediglich eine Python-Laufzeitumgebung voraussetzt.

6.2 Literaturverzeichnis

NS-3 Tutorial, ns-developers@isi.edu, http://www.nsnam.org/docs/release/tutorial.pdf 10
http://de.wikipedia.org/wiki/Real-time_Transport_Protocol, 01.04.09 17
http://mmep.ite.fh-wiesbaden.de/info/rtp.htm, 28.04.09 18
http://de.wikipedia.org/wiki/Real-time_Transport_Protocol, 01.04.09 18
http://de.wikipedia.org/wiki/Real-time_Transport_Protocol, 01.04.09 18
http://www.cisco.com/en/US/tech/tk652/tk698/technologies_tech_note09186a0080094ae2.shtml, 02.04.09 20
Key Material and Random Numbers, Andreas Steffen, Seite 8, 19.02.2009 20
Analysis of the Linux Random Number Generator, Gutterman & Pinkas, Kapitel 2 ff. 21
Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators, Bundesamt für Sicherheit in der Informationstechnik, http://www.bsi.de/zertifiz/zert/interpr/ais20e.pdf .. 22
Der CHI-Quadrat Test, Blenkers & Becker, http://www.audicon.net/downloads/artikel/Chi-Quadrat-Test.pdf 22
A statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST Special Publication 800-22, Andrew Rukhin et al., Kapitel 2.1, http://www.random.org/statistics/SP800-22b.pdf 22
A statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST Special Publication 800-22, Andrew Rukhin et al., Kapitel 2.3, http://www.random.org/statistics/SP800-22b.pdf 22

6.3 Abbildungsverzeichnis

6.4 Tabellenverzeichnis

7 Anhang

7.1 User Manual

7.1.1 Requirements and preconditions

TraGiC is able to work along with every network built with NS-3. The TrafficGenerator is a simple NS-3::Application which can be installed on any node, so you can use your very own NS-3 network and just add TraGiC to it.

You can use the sample code of the tutorial to NS-3[18] to test TraGiC or take one of the numerous examples. Alternatively, you can use the fully working "Hello World" example in the last chapter.

7.1.2 TraGiC Simulation

7.1.2.1 Setting up an RTP network

To set up a network, TraGiC follows the NS-3 style setup with internet stack helpers. After creating a group with 10 nodes we can install the RTP stack. Net devices are allocated as usual in NS-3.

```
        // setting up network
        CsmaHelper csmaHelper;
        RtpStackHelper rtpHelper;
        Ipv4AddressHelper ipv4Helper;

        NodeContainer group;
        group.Create(10);
        rtpHelper.Install(group);
        NetDeviceContainer netDevices = csmaHelper.Install(group);
        ipv4Helper.SetBase("10.0.0.0", "255.255.0.0");
        ipv4Helper.Assign(netDevices);
```

7.1.2.2 Enable PCAP recording

To enable PCAP recording, we usually use the following code part:

```
        // enable reporting
        CsmaHelper::EnablePcap("MyPCAP", groups.at(0).Get(0)->GetDevice(0), true);
```

This will enable reporting on the first node in the first network ("groups" is a vector of node containers). If you have more than one network, you have to add this line for each network. It will produce a PCAP file with all conversations in one network ("true" stands for promiscuous mode).

Needless to say that you can handle the way of recording the network data yourself; this is just an example how TraGiC was used in development.

7.1.2.3 Generating VoIP streams

The primary use of TraGiC is to create VoIP streams between nodes. TraGiC provides an NS-3 style helper class which facilitates the process of setting up a VoIP traffic generator. The main functionalities when using the helper with a VoIP coordinator are:

- Bidirectional connections: Calls, once set up, will stream RTP data in both directions to make the simulation more realistic.
- Call management: Each node is controlled by the coordinator. The coordinator is asked by the nodes where to call next and will distribute the calls according to a given distribution table, which controls incoming and outgoing calls.

[18] http://www.nsnam.org/docs/release/tutorial.html, 06.06.09

- A node can only communicate with one other node at the same time.
- Erlang statistics: The coordinator is also able to print a table of all networks (node containers) and show their incoming and outgoing Erlang values similar to the distribution table.

7.1.2.3.1 Code

The following code is needed to install the VoIP Traffic Generator on an existing network.

First of all, we need an NS-3 network. Usually this is already set up and is also discussed in the NS-3 manual, so we use this one-line utility method which sets up a CSMA network with two groups and three nodes each.

```
        // setting up network
        vector<NodeContainer> groups = Util::createCsmaNetwork(list_of(3)(3));
```

Afterwards, we need a coordinator. The parameters in the constructor are the mean call duration and the mean call offset time (before the node asks for a new target). Be aware that these values are determined via a negative exponential distribution, which means that lower values are much more probable.

```
        // create coordinator
        Ptr<VoipTrafficCoordinator> coordinator =
                        Create<VoipTrafficCoordinator> (Seconds(300), Seconds(10));
```

The coordinator also needs a distribution table which defines the traffic distribution. Assuming we have two networks A and B and want to distribute the traffic in A equally and the traffic in B that 2/3 of all nodes connect to A and 1/3 to B. The table would look as following:

	A	B
A	1	1
B	2	1

Table 1: Distribution Table Example

Note that the absolute values are only important per row (the distribution in one network) and floating point numbers are allowed. Zero values can be omitted; there could also be a C network in the table which does not have any incoming or outgoing streams.

The code to the table above would be, assuming that network A is at position 0 in the vector of node containers:

```
        // define distribution table
        map<NodeContainer, double> trafficGroup0;
        map<NodeContainer, double> trafficGroup1;
        trafficGroup0[groups.at(0)] = 1;
        trafficGroup0[groups.at(1)] = 1;
        trafficGroup1[groups.at(0)] = 2;
        trafficGroup1[groups.at(1)] = 1;
        coordinator->addNetworkTraffic(groups.at(0), trafficGroup0);
        coordinator->addNetworkTraffic(groups.at(1), trafficGroup1);
```

The only thing left is to install the generator and the sink (to accept incoming traffic) on each node. We can use the media helper here, because all specialized VoIP code is included in the coordinator.

```
        // installing generator and sink on node containers
        MediaTrafficGeneratorHelper mediaHelper(coordinator);
        PacketSinkHelper sinkHelper("NS-3::UdpSocketFactory", InetSocketAddress(
                        Ipv4Address::GetAny(), MediaTrafficCoordinator::PORT));
        ApplicationContainer applications;
        applications.Add(mediaHelper.Install(groups.at(0)));
        applications.Add(mediaHelper.Install(groups.at(1)));
```

```
        applications.Add(sinkHelper.Install(groups.at(0)));
        applications.Add(sinkHelper.Install(groups.at(1)));
```

And finally schedule the applications to start randomly between 0 and 0.001 seconds. The loop is necessary because the NS-3 ApplicationContainer class does not support scheduling the application start randomly, but the Application class does.

```
        // schedule applications
        for (ApplicationContainer::Iterator it = applications.Begin(); it
                        != applications.End(); ++it) {
                (*it)->Start(UniformVariable(0, 0.001));
        }
        applications.Stop(Seconds(300));
```

Scheduling the applications randomly is done to create a more realistic scenario. If all applications start at the same time (as with applications.Start(Seconds(0))), the nodes in the first network will ask for a call first and thus falsify the simulation. When running the simulation for a longer time, the transient effect will be reduced.

Afterwards the simulator can be started as shown in the last chapter in the code example.

```
        // run simulation
        Simulator::Run();
        Simulator::Destroy();
```

To print out the Erlang values, you can call the following method at any time in the simulation. It will return a DistributionMatrix which can be printed to any stream (file stream, console output) via the overloaded stream operator.

```
        coordinator->getErlangStatistics();
```

The values in the table are calculated from the beginning of the simulation until now. It is recommended to call this method between Simulator::Run() and Simulator::Destroy() to include the whole simulation.

7.1.2.4 Generating general media streams

In case you just want to stream random RTP data and do not care about VoIP or calls at all, you can use the helper class with another coordinator. The following features differ from the VoIP functionalities:

- Unidirectional connections: RTP data will only be streamed in one direction, like a video stream.
- Distribution table: As in the VoIP traffic coordinator, the media traffic coordinator uses a distribution table to determine the next node to be called. However, it does not assure that only one connection per node is active, since one can receive or send multiple media streams at the same time.

7.1.2.4.1 Code

The code to create this kind of traffic generator is easy, not to say exactly the same as the one above but one part. The VoipTrafficCoordinator has to be changed into MediaTrafficCoordinator.

```
        // create coordinator
        Ptr<MediaTrafficCoordinator> coordinator =
                        Create<MediaTrafficCoordinator> (Seconds(300), Seconds(10));
```

7.1.2.5 Using another Codec

The default codec used in both the VoIP and the media generator is G711 PCMA. To change the codec, simply put the following line after creating the media helper. There are a few given media codecs which can be accessed as a static variable in the class Codec.

```
        mediaHelper.setCodec(Codec.G729);
```

The codecs available in TraGiC are, according to the Cisco specification[19]:

- G.711 PCMA
- G.711 PCMU
- G.723 (5.3 kbps)
- G.723 (6.3 kbps)
- G.726 (24 kbps)
- G.726 (32 kbps)
- G.728
- G.729

To create an own codec, use the constructor of the codec class:

```
        mediaHelper.setCodec(Codec(10, MilliSeconds(5), 15, 6));
```

The parameters are the sample size in bytes, the sample interval, the RTP payload type and the factor of the samples (default 1). These values can be looked up in the RFC or other sources on the net (The shown codec above is G728).

7.1.2.6 Using another strategy

If you want to change the way the traffic generator chooses its next target address, packet offset or packet size, you need to change the strategy the generator uses. The following code uses an RTP socket and a random deviation generator strategy which chooses its values randomly. The average packet offset is 10 Milliseconds.

```
        // setting the custom strategy
        TrafficGeneratorHelper helper;
        helper.setSocketFactoryType(TypeId::LookupByName("NS-3::RtpSocketFactory"));
        Ptr<RandomDeviationGeneratorStrategy> strategy = Create<
                        RandomDeviationGeneratorStrategy> (MilliSeconds(10));
```

Now we need to add the address probabilities, because the random deviation strategy uses these values to determine the next target address. Of course, if your own strategy uses different methods to look up the addresses, this part can be ignored. Here you set the custom strategy values and assign it to the helper.

```
        // adding address probabilities
        strategy->setAddressProbability(InetSocketAddress(
                        groups.at(0).Get(0)->GetObject<Ipv4> ()->GetAddress(1), 1234), 0.25);
        strategy->setAddressProbability(InetSocketAddress(
                        groups.at(0).Get(1)->GetObject<Ipv4> ()->GetAddress(1), 1234), 0.25);
        strategy->setAddressProbability(InetSocketAddress(
                        groups.at(1).Get(0)->GetObject<Ipv4> ()->GetAddress(1), 1234), 0.25);
        strategy->setAddressProbability(InetSocketAddress(
                        groups.at(1).Get(1)->GetObject<Ipv4> ()->GetAddress(1), 1234), 0.25);
        helper.setStrategy(strategy);
```

[19] http://www.cisco.com/en/US/tech/tk652/tk698/technologies_tech_note09186a0080094ae2.shtml, 06.06.09

Note: 1234 is the standard port for RTP traffic used in TraGiC.

Afterwards we have to install the generators and packet sinks on the nodes as before.

```
        // installing generator and sink on node containers
        PacketSinkHelper sinkHelper("NS-3::UdpSocketFactory", InetSocketAddress(
                        Ipv4Address::GetAny(), 1234));
        ApplicationContainer applications;
        applications.Add(helper.Install(groups.at(0)));
        applications.Add(helper.Install(groups.at(1)));
        applications.Add(sinkHelper.Install(groups.at(0)));
        applications.Add(sinkHelper.Install(groups.at(1)));
```

Scheduling/Starting the applications works as above.

7.1.2.7 Using another protocol

To use another protocol, we just have to change one line in the code above. The TrafficGeneratorHelper needs another type of socket:

```
        helper.setSocketFactoryType(TypeId::LookupByName("NS-3::RtpSocketFactory"));
```

Currently, only RTP is implemented, so please consult the developer manual to add support for another protocol.

Be aware that you also have to differ between UDP and TCP when installing the packet sink on the nodes. The other part of the code works the same way as above.

7.1.3 Visualization

NS-3 does not provide any visualization. Instead, all packets are printed out directly into console or to a standardized PCAP file, where each packet is stored as binary data. Because of the PCAP standard there are many other tools to visualize the data. The following applications have been chosen for TraGiC, however there are many others and this list is not complete.

7.1.3.1 Wireshark[20]

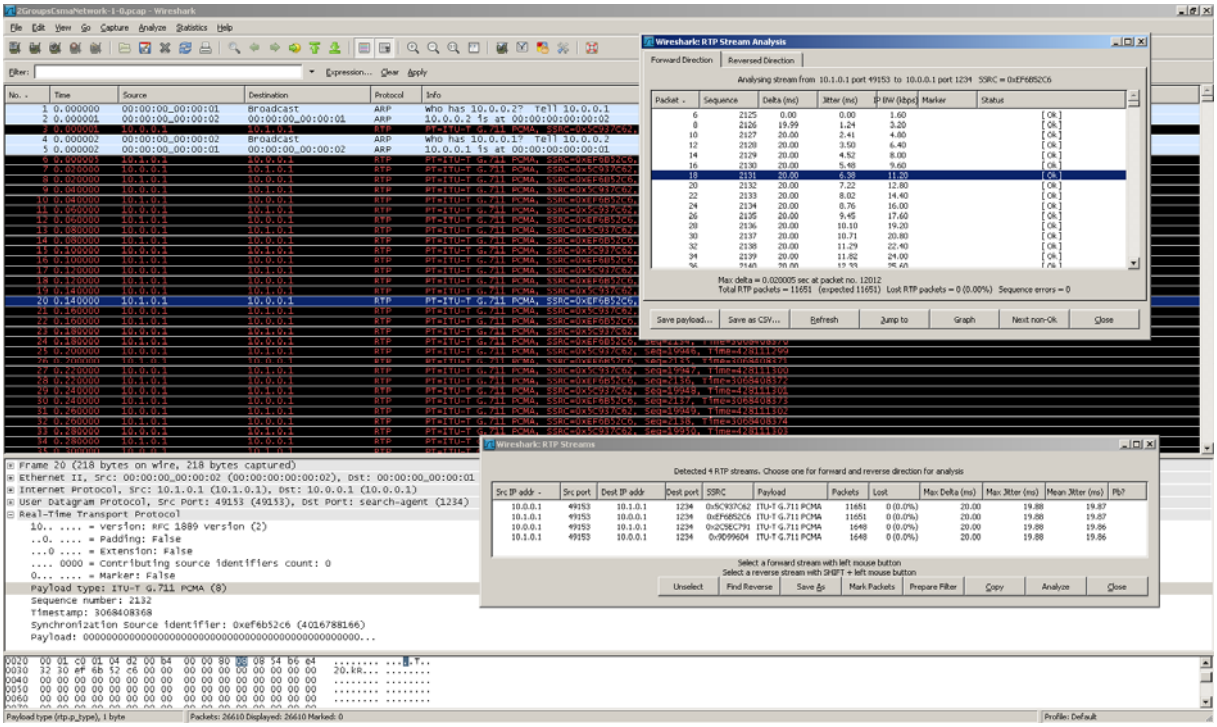


Figure 1: Wireshark

Wireshark (former Ethereal) is one of the best known tools to sniff networks and create or analyze PCAP files. The packets are listed in a simple but effective view and the program provides many filter- and analyzing methods to investigate the recorded traffic.

PCAP files created by TraGiC can be read by Wireshark and will be recognized as UDP traffic. This is because TraGiC does not set up the call, and thus Wireshark will not recognize the start of an RTP stream and will ignore the RTP header at first. This is because RTP is just a kind of a UDP wrapper and belongs to the same OSI layer as TCP/UDP. Clicking the right mouse button on a UDP packet and choosing “Decode As…” → “RTP” will try to decode all UDP packets in the file. This also enables stream analysis which is available via “Statistics” → “RTP”.

Wireshark is available for both Linux and Windows and can be installed easily. Opening a PCAP file with Wireshark will automatically show the packet list.

[20] http://www.wireshark.org/, 19.05.09

7.1.3.2 INAV[21]

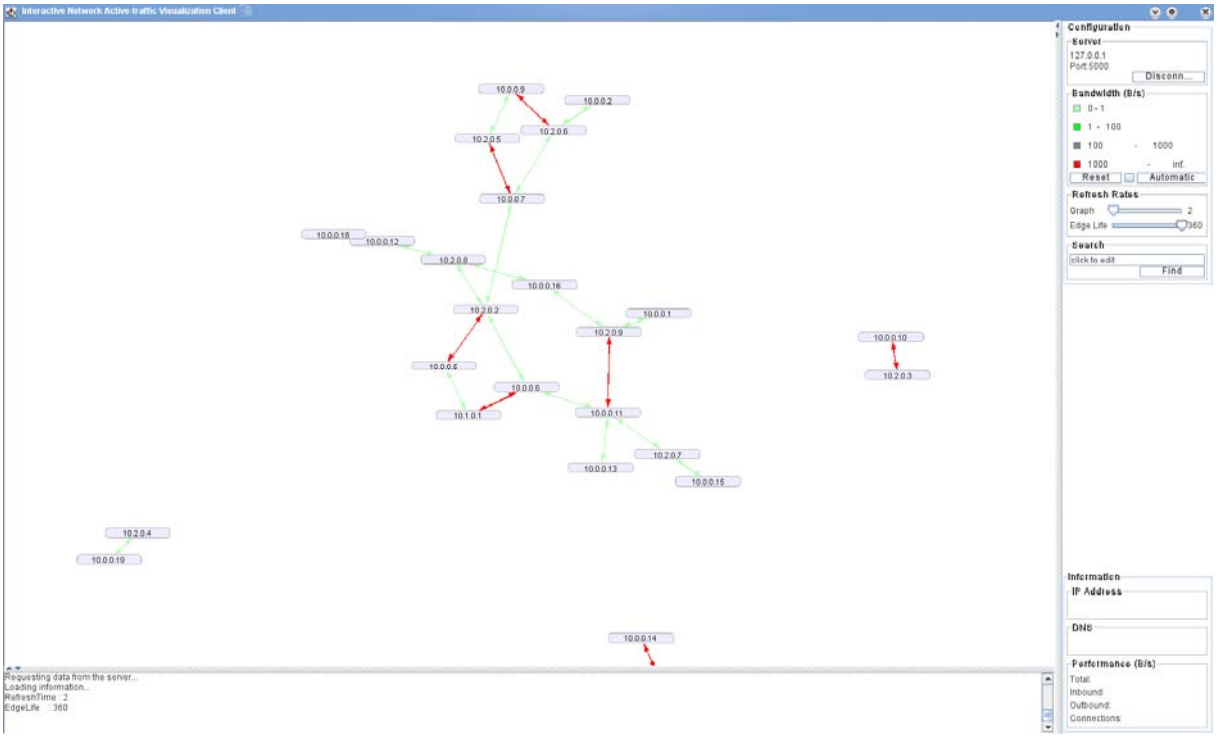


Figure 2: INAV client

INAV is a Linux client/server program where the server is responsible for all analysis and the client is solely for visualization use. INAV could also be started as a real-time sniffer like Wireshark, but is much more visual oriented instead of exact packet analysis.

When INAV is installed, the PCAP file can be played with the following command:

```
# ./inavd –f pcapfile.pcap
```

And the client is started with:

```
# java –jar INAV-0.15.jar
```

After the client has connected to the server, it will show the PCAP playing. Nodes are displayed as IP addresses which will appear as they start or receive a stream. Traffic between them is showed as arrows where active connections are painted in red. Older connections have a lighter color and will disappear after a configurable time (“Edge Life”).

[21] http://inav.scaparra.com/, 19.05.09

7.1.3.3 SMART[22]

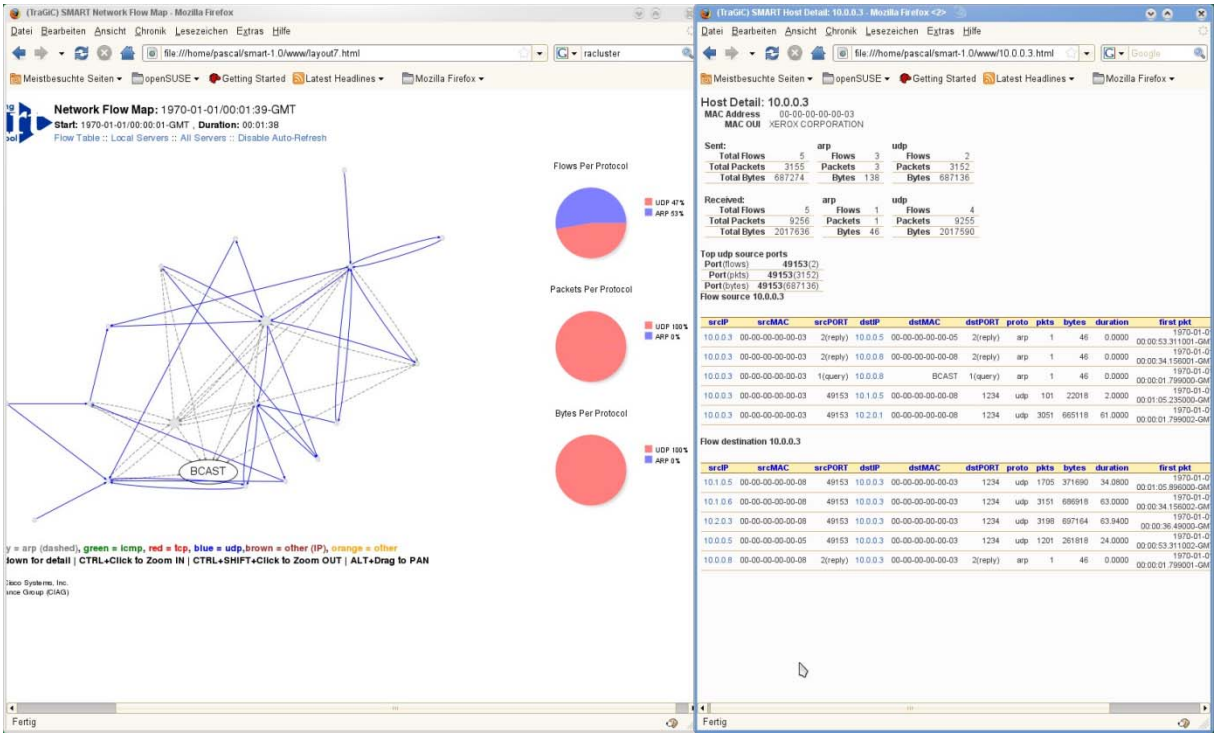


Figure 3: SMART

SMART (short for Safe Mapping and Reporting Tool) is a network flow analysis tool for Linux (Windows is supported, but does not work in the current version). The network can be observed in real-time and analyzed by protocol flow (one can see how much of the traffic was ARP, RTP etc.). SMART also supports PCAP reading. Single nodes are displayed with detailed information and connections from and to nodes are shown in a chronological order. In the graphical overview, the nodes are connected to the other nodes to which they had a connection in the analyzed time span.

SMART can be started with the following command. After that the output can be viewed with an SVG compatible browser. Navigating to the web server of SMART will show the graphical overview and the network flow analysis.

```
# ./smart.pl –r pcapfile.pcap
```

7.1.3.4 Other visualizations

- Flowtime[23]: Perl script with chronological overview of the traffic
- AfterGlow[24]: Script collection to display network graphs (and others)

7.1.4 Code example / Hello World

The following is a full code example of a simulation. The main method enables packet metadata which starts the collection of metadata before the output formats are known. This allows specifying the desired format (PCAP, file, etc) at any given point in the simulation. Furthermore, the simulation random seed is set, so we can repeat the simulation at any time and get exactly the same results.

```
int TraGiCSimulator::main(const vector<string> &args) {
```

[22] http://safemap.sourceforge.net/, 19.05.09
[23] http://thnetos.wordpress.com/2008/01/24/flowtime-create-a-timeline-for-packet-flow/, 19.05.09
[24] http://afterglow.sourceforge.net/, 19.05.09

```
        PacketMetadata::Enable();
        SeedManager::SetSeed(0xf18b);
        simulateVoipTrafficGenerator();
        return 0;
}
```

Afterwards we set up the network as described above, install all necessary applications and start the simulator. In the end, an Erlang statistic is printed out.

```
void TraGiCSimulator::simulateVoipTrafficGenerator() {
        // setting up network
        CsmaHelper csmaHelper;
        RtpStackHelper rtpHelper;
        Ipv4AddressHelper ipv4Helper;

        Ptr<Node> routerNode = CreateObject<Node> ();
        NodeContainer groupA;
        NodeContainer groupB;
        groupA.Create(10);
        groupB.Create(10);
        rtpHelper.Install(groupA);
        rtpHelper.Install(groupB);
        rtpHelper.Install(routerNode);
        NodeContainer groupAWithRouter(groupA);
        groupAWithRouter.Add(routerNode);
        NodeContainer groupBWithRouter(groupB);
        groupBWithRouter.Add(routerNode);
        NetDeviceContainer netDevicesA = csmaHelper.Install(groupAWithRouter);
        NetDeviceContainer netDevicesB = csmaHelper.Install(groupBWithRouter);
        ipv4Helper.SetBase("10.0.0.0", "255.255.0.0");
        ipv4Helper.Assign(netDevicesA);
        ipv4Helper.SetBase("10.1.0.0", "255.255.0.0");
        ipv4Helper.Assign(netDevicesB);

        GlobalRouteManager::PopulateRoutingTables();

        // create coordinator
        Ptr<VoipTrafficCoordinator> coordinator = Create<VoipTrafficCoordinator> (
                        Seconds(30), Seconds(10));

        // define distribution table
        map<NodeContainer, double> trafficGroupA;
        map<NodeContainer, double> trafficGroupB;
        trafficGroupA[groupA] = 0.5;
        trafficGroupA[groupB] = 1;
        trafficGroupB[groupA] = 1;
        trafficGroupB[groupB] = 0.5;
        coordinator->addNetworkTraffic(groupA, trafficGroupA);
        coordinator->addNetworkTraffic(groupB, trafficGroupB);

        // installing generator and sink on node containers
        MediaTrafficGeneratorHelper mediaHelper(coordinator);
        PacketSinkHelper sinkHelper("NS-3::UdpSocketFactory", InetSocketAddress(
                        Ipv4Address::GetAny(), MediaTrafficCoordinator::PORT));
        ApplicationContainer applications;
        applications.Add(mediaHelper.Install(groupA));
        applications.Add(mediaHelper.Install(groupB));
        applications.Add(sinkHelper.Install(groupA));
        applications.Add(sinkHelper.Install(groupB));

        // enable reporting
        CsmaHelper::EnablePcap("HelloTraGiC", routerNode->GetDevice(0), true);

        // schedule applications
        for (ApplicationContainer::Iterator it = applications.Begin(); it
                        != applications.End(); ++it) {
                (*it)->Start(UniformVariable(0, 0.001));
        }
        applications.Stop(Seconds(300));

        // run simulation
        Simulator::Run();
        clog << coordinator->getErlangStatistics() << endl;
        Simulator::Destroy();
}
```

Notes:

- The router node is a special node which is available in both networks A and B. Because the internet stack helper class does not check if it is already installed and will crash when installed twice, we need to install the RTP stack separately on the router node.
- Populating the routing tables is vital to the simulation, or else there will be no routing.
- Be careful to not add the router node to the coordinator (distribution table). TraGiC does not differ when a node has more than one IP address.

7.2 Developer Manual

7.2.1 Preparing the development environment

7.2.1.1 Overview

As TraGiC represents an addition to the actual NS-3 library, it is not necessary to compile it directly into the NS-3 library itself. Instead, we decided to compile it as a separate library and link it against the precompiled NS-3 library, saving us the effort of recompiling and relinking the complete NS-3 library on every build. This chapter will inform you on how to create a development environment, including a compiler and an IDE, allowing you to use and extend NS-3 in your programs.

7.2.1.2 Compile NS-3 library

Even though the NS-3 library is compatible with various platforms and compilers, it does not provide any Makefiles for the popular GNU make automatic build tool. Instead, the Python-based waf [25]build system is used, which itself is compatible to the GNU C++ compiler g++. Since version 3.4, NS-3 also provides MinGW platform support, so compiling and running NS-3 programs on Windows is easily feasible. While the NS-3 tutorial[26] basically provides solid installation and compilation instructions, this manual states some additional information concerning certain issues during the initial compilation of NS-3.

7.2.1.2.1 Install Python[27]

The waf build system requires a python interpreter installed on the respective system in order to build the NS-3 library. Please note that waf exhibits severe compatibility issues with newer python versions than 2.5 and we strongly recommend installing only this version and adding it to your PATH system variable.

7.2.1.2.2 Using waf

The NS-3 tutorial instructs you to download and install waf and create a system variable called WAFDIR pointing to your installation directory. Please do not follow these instructions, since the NS-3 waf build files do not cope well with varying waf versions. Instead, simply use the waf Python script included in your NS-3 build, as this version is assured to comply with the given build files.

7.2.1.3 Install toolchain

To begin developing C++ programs, a development toolchain including a C++ compiler is necessary. NS-3 renders itself compatible to various platforms and compilers and depending on your choice, different compiler commands and programs must be used to build your programs. In this manual,

[25] waf, The flexible build system, http://code.google.com/p/waf/
[26] ns-3 Tutorial (html version), Chapter 3: Getting started, http://www.nsnam.org/docs/release/tutorial.html#SEC11
[27] Python, Python Programming Language, http://www.python.org/

only the GNU make system and its g++ compiler will be examined. For Windows, we recommend using the MinGW toolchain, while on Linux any GNU compatible toolchain may be used. We do not recommend using cygwin on Windows, as even though the programs compile fine and work flawlessly, cygwin interprets some linker statements generated by the NS-3 build as erroneous and thus displays many unnecessary error messages.

7.2.1.4 Install IDE

For any but the most trivial programs, we highly recommend using a graphical IDE supporting your development work, as programming with NS-3 is neither trivial nor – in some cases – intuitive. For the GNU utils compiler, eclipse [28]is a suitable choice. Please be aware that eclipse expects a g++ compiler to be found in your system PATH variable. Otherwise, compilation will fail. On both Windows and Linux, assure that your compiler of choice is the first one to be found by eclipse in your system PATH. Otherwise, eclipse and GNU work flawlessly together and require no further configuration. Only when using MinGW on Windows, some additional settings must be taken, as explained in the following.

7.2.1.4.1 Eclipse with MinGW

For allowing eclipse to use MinGW and all its features properly, these auxiliary project settings must be taken:

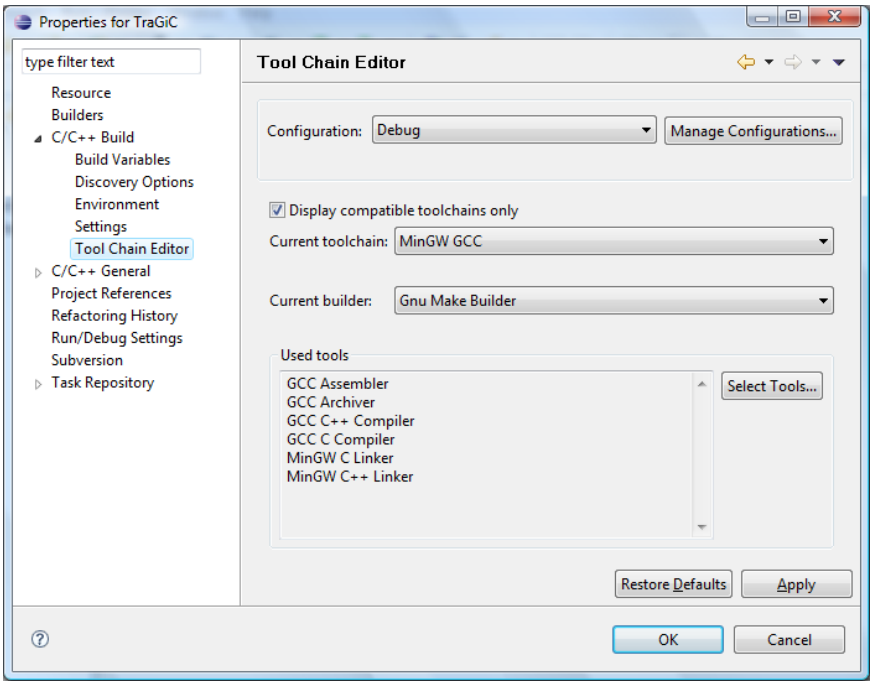


Figure 4: Enabling GNU Make Builder in Eclipse

First, select MinGW GCC as your current toolchain and mark “Gnu Make Builder” as your current builder.

[28] Eclipse IDE for C/C++ Developers, http://www.eclipse.org/downloads/

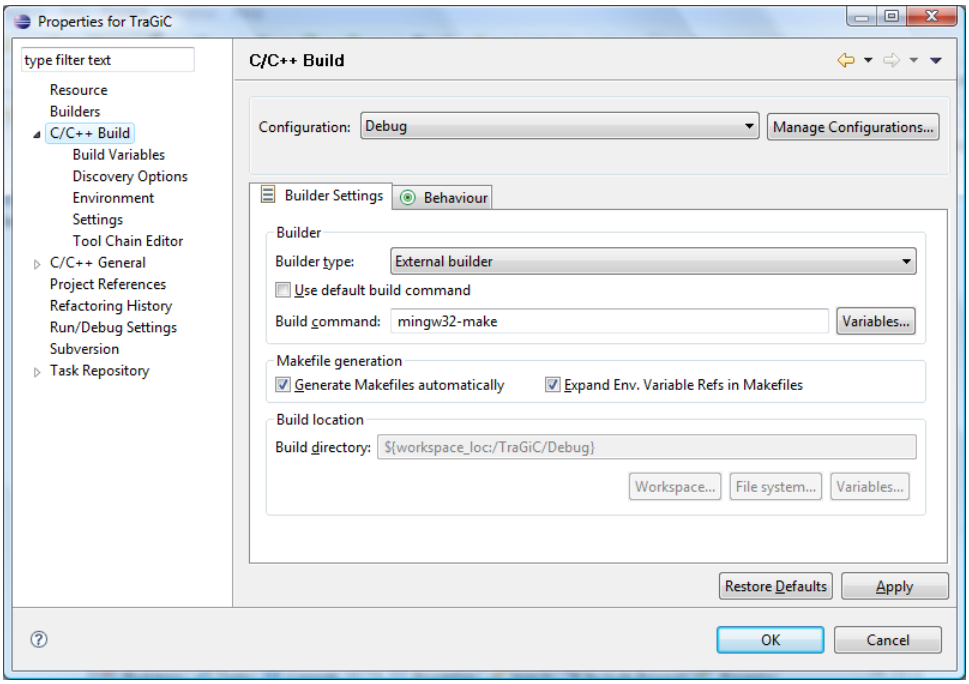


Figure 5: Specifying mingw32-make as build command

Afterwards, specify “mingw32-make” instead of simply “make” as your build command.

Adjust these parameters on every project you create and MinGW C++ builds should run flawlessly on your computer.

7.2.2 Architecture and design of TraGiC

7.2.2.1 Design and Who's Who in TraGiC

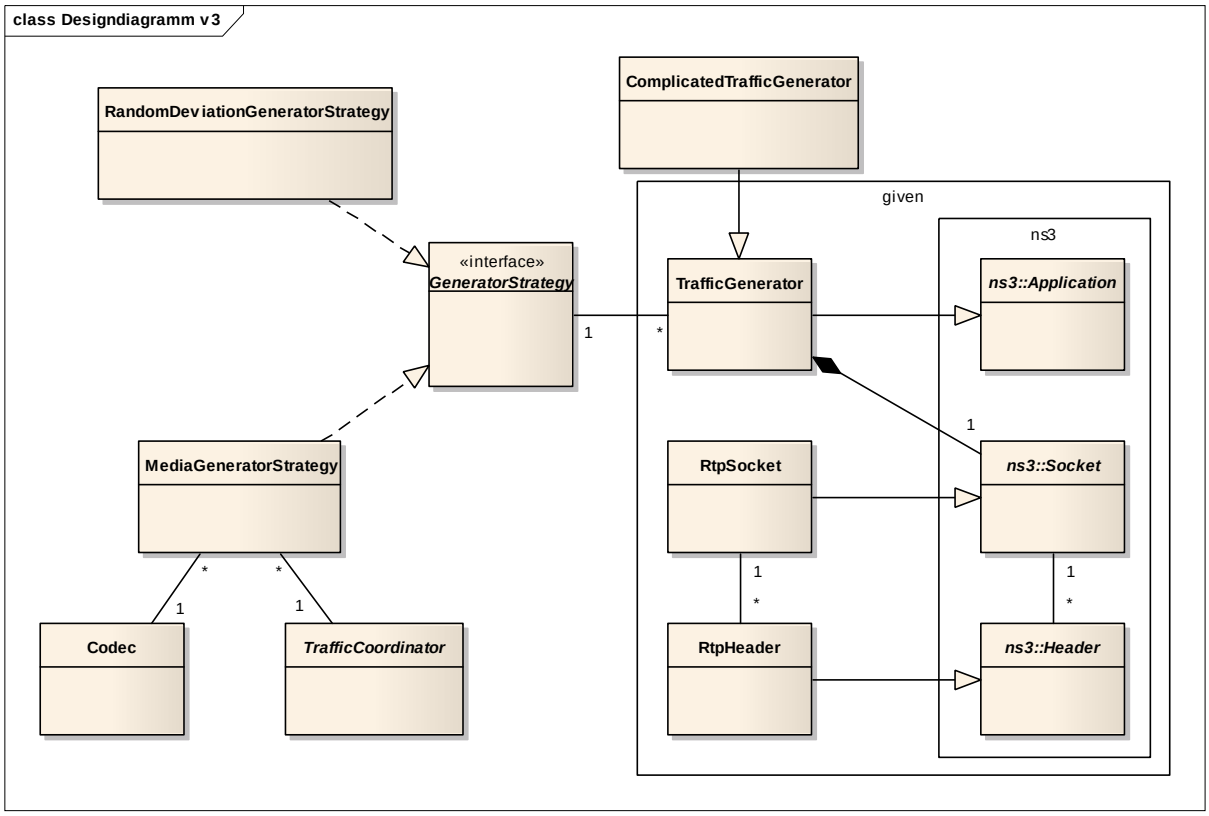


Figure 6: Design Diagram

The Design of TraGiC is simple to understand and makes it easy to add more functionality to the Traffic Generator. It can be divided into three main groups: The "standalone" RTP part with the classes RtpSocket and RtpHeader, the TrafficGenerator class itself and the strategy part including the interface GeneratorStrategy and the classes MediaGeneratorStrategy, Codec and TrafficCoordinator.

7.2.2.1.1 RTP

This part of TraGiC is solely the implementation of the RTP protocol. According to NS-3, one needs to create a socket class and a header class to implement a new protocol. Additionally, due to the NS-3 design structure, a SocketFactory instantiating the sockets must be provided for each protocol separately, resulting in TraGiC also implementing an RtpSocketFactory.

7.2.2.1.1.1 RtpSocket

The RtpSocket class is mainly a wrapper to a UdpSocket class with added RTP functionality such as setting the payload type of RTP (which is needed in the header). Its functionality is split into its interface, the RtpSocket class, and the implementation RtpSocketImpl. This is a design guideline by NS-3, allowing the seamless exchange of different Protocol implementations.

7.2.2.1.1.2 RtpHeader

This class represents the header of an RTP data packet. Header specific values such as timestamp or sequence number are initialized on creating the header. Each time a packet is sent via the RTP socket, a new header will be generated and added to the packet.

7.2.2.1.2 Traffic Generator

The traffic generator is a rather simple class which has a socket and a strategy. It will ask the strategy for the three parameters packet size, packet time offset and next target address and will send the packet via the specified socket.

7.2.2.1.3 GeneratorStrategy

As written before, this class is responsible for determining the packet size, packet time offset and the next target address. Each time a packet is sent, the strategy will be asked for these parameters. Thus the strategy could randomly select from a pool of addresses, always return the same packet size and vary between one and twenty milliseconds in packet time offset – or do something completely different.

The default strategy in TraGiC is a MediaGeneratorStrategy, which usually works together with an RtpSocket.

7.2.2.1.3.1 Codec

In a MediaGeneratorStrategy, a standardized codec is used to determine packet size and time offset, according to the Cisco specification[29]. You can simply add a codec to the list in the header file of the Codec class and initialize it in the source file.

7.2.2.1.3.2 Coordinator

A MediaGeneratorStrategy needs a coordinator to get the next target address. The default MediaCoordinator uses a given traffic distribution matrix to calculate the next address. In addition the VoipTrafficCoordinator has a few methods to manage the calls between nodes and makes sure that a node is talking only to one other node.

These values are stored in a global coordinator to avoid many different strategies. Your own strategy does not necessarily need a coordinator; it's up to you what other classes you will need.

7.2.2.2 Helper classes

Hidden in the design diagram are the topology helper classes[30] which represent a standard practice in NS-3.

7.2.2.2.1 RtpStackHelper

The RtpStackHelper is basically an improvement to the NS-3 InternetStackHelper[31]. It applies the latter on the selected Node or NodeContainer and adds the RtpSocketFactory using the NS-3 object aggregation interface.

7.2.2.2.2 ApplicationHelper

To facilitate installing applications, we've created an application helper class. Instead of programming the whole helpers for each application again, our generator-helpers inherit from ApplicationHelper and use the methods provided.

TrafficGeneratorHelper is a direct subclass and will install a TrafficGenerator when a socket and a strategy are provided. The MediaTrafficGeneratorHelper, which inherits from the TrafficGeneratorHelper, only needs a Codec and a Coordinator to install all necessary components.

---

[29] http://www.cisco.com/en/US/tech/tk652/tk698/technologies_tech_note09186a0080094ae2.shtml, 07.06.09
[30] http://www.nsnam.org/docs/release/tutorial.html#SEC26, 07.06.09
[31] http://www.nsnam.org/docs/release/tutorial.html#SEC37, 07.06.09

Please consult the user manual to see how to use these helpers.

7.2.2.3 Sequence Diagram

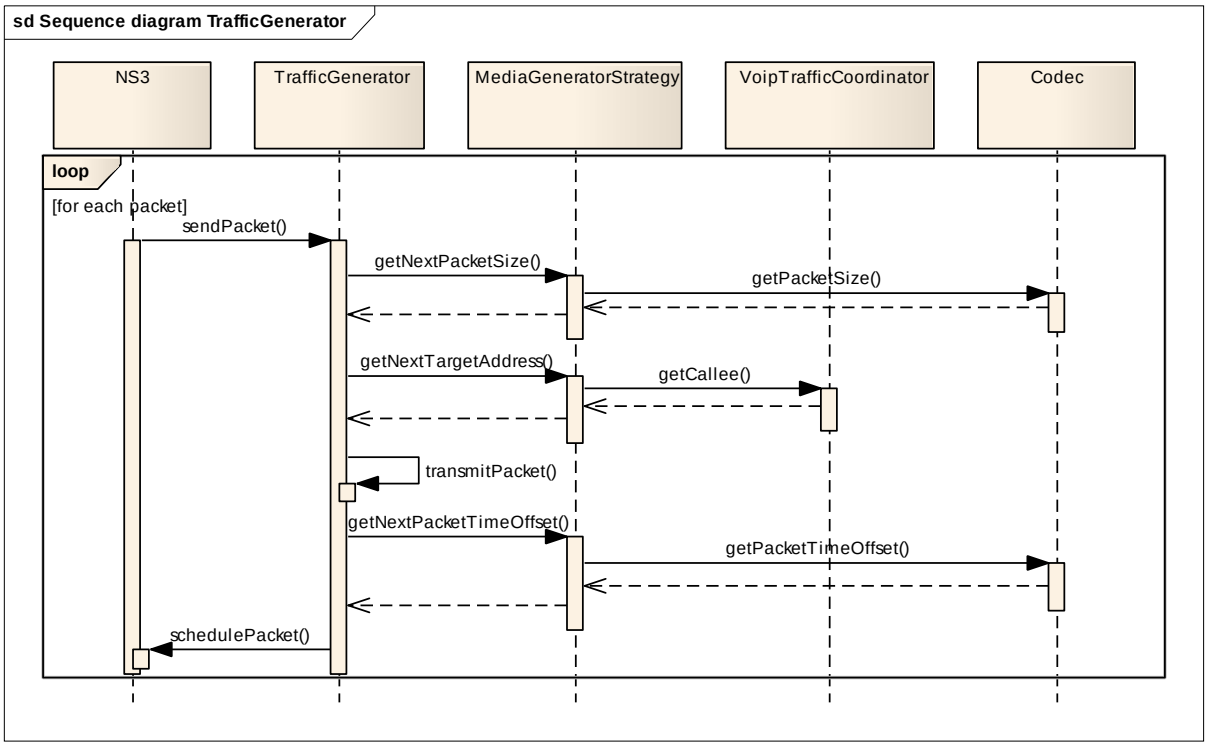


Figure 7: Sequence diagram of traffic generator and strategy

This diagram shows how TraGiC effectively works. The TrafficGenerator asks his strategy for the values for each packet. Exactly how the strategy gathers these parameters is not prescribed, but in the case of the media strategy it will use a coordinator (here: VoIP) and a codec.

The sending of the packet itself happens in the transmitPacket()-Method which will be illustrated in detail in the next chapter. Please note that transmitPacket() is not an actual method in the program. It is only mentioned in the diagram for illustrating that at this point in the diagram, the packet transmission progress described in the next diagram is applied.

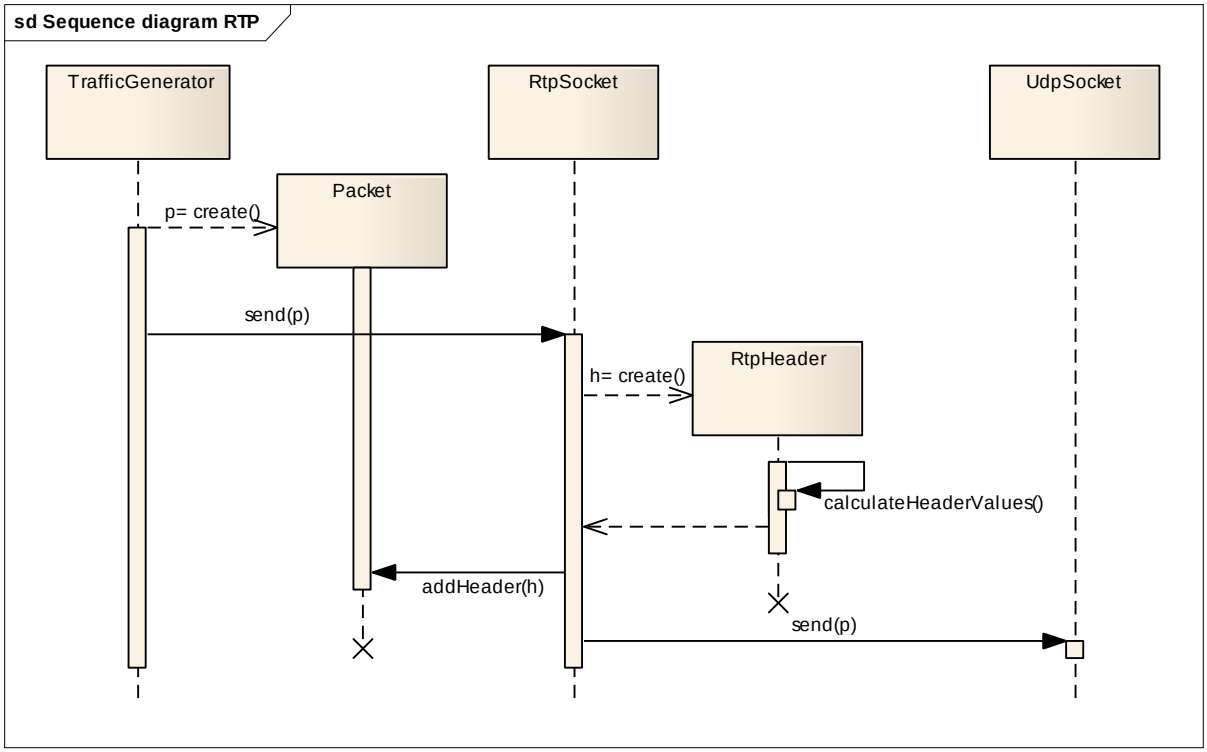


Figure 8: Sequence diagram RTP

The packet header is created, filled with the correct values and added to the packet. Afterwards, it will be sent via the wrapped UdpSocket.