DiSIT, Computer Science Institute
Università del Piemonte Orientale “A. Avogadro”
Viale Teresa Michel 11, 15121 Alessandria
http://www.di.unipmn.it

UNIVERSITÀ DEL PIEMONTE ORIENTALE

Sviluppo in OMNeT++ di modelli di cyber attacchi a risorse energetiche distribuite

D. Savarro (davide.savarro@uniupo.it)

TECHNICAL REPORT TR-INF-2023-12-02-UNIPMN
(December 2023)

Research Technical Reports published by DiSIT, Computer Science Institute, Università del Piemonte Orientale are available via WWW at URL `http://www.di.unipmn.it/`. Plain-text abstracts organized by year are available in the directory

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2023-01 *NRTS: A Client-Server architecture for supporting data recording, transmission and evaluation of multidisciplinary teams during the neonatal resuscitation simulation scenario*, M. Canonico, S. Montani, M. Striani, April 2023.

2022-01 *A modular infrastructure for the validation of cyberattack detection systems*, D. Cerotti, D. Codetta Raiteri, G. Dondossola, L. Egidi, G. Franceschinis, L. Portinale, R. Terruggia, May 2022.

2021-01 *General composition for Symmetric Net arc functions with applications*, L. Capra, M. De Pierro, G. Franceschinis, May 2021.

2020-05 *Weighted defeasible knowledge bases and a multipreference semantics for a deep neural network model*, L. Giordano, D. Theseider Dupré, December 2020.

2020-04 *A reconstruction of multipreference closure*, L. Giordano, V. Gliozzi, September 2020.

2020-03 *A framework for a modular multi-concept lexicographic closure semantics*, L. Giordano, D. Theseider Dupré, September 2020.

2020-02 *On a plausible concept-wise multipreference semantics and its relations with self-organising maps*, L. Giordano, V. Gliozzi, D. Theseider Dupré, September 2020.

2020-01 *Reasoning about exceptions in ontologies: from the lexicographic closure to the skeptical closure*, L. Giordano, V. Gliozzi, March 2020.

2019-05 *Renvoi in Private International Law: a Formalization with Modal Contexts*, L. Giordano, B. Matteo, K. Satoh, October 2019.

2019-04 *UML class diagrams supporting formalism definition in the Draw-Net Modeling System*, D. Codetta Raiteri, July 2019.

2019-03 *Tracing and preventing sharing and mutation*, P. Giannini, M. Servetto, E. Zucca, July 2019.

2019-02 *The Android Forensics Automator (AnForA): a tool for the Automated Forensic Analysis of Android Applications*, C. Anglano, M. Canonico, M. Guazzone, June 2019.

2019-01 *Deriving Symbolic and Parametric Structural Relations in Symmetric Nets: Focus on Composition Operator*, L. Capra, M. De Pierro, G. Franceschinis, March 2019.

2018-03 *Deriving Symbolic Ordinary Differential Equations from Stochastic Symmetric Nets without Unfolding*, M. Beccuti, L. Capra, M. De Pierro, G. Franceschinis, S. Pernice, July 2018.

2018-02 *Power (set) Description Logic*, L. Giordano, A. Policriti, February 2018.

2018-01 *A Game-Theoretic Approach to Coalition Formation in Fog Provider Federations (Extended Version)*, C. Anglano, M. Canonico, P. Castagno, M. Guazzone, M. Sereno, February 2018.

Sviluppo in OMNeT++ di modelli di cyber attacchi a risorse energetiche distribuite

Davide Savarro

Dicembre 2023

Sommario

In un mondo in cui le comuni infrastrutture centralizzate di produzione e distribuzione dell'energia elettrica diventano gradualmente obsolete e le energie rinnovabili acquisiscono sempre più popolarità e finanziamenti, si rivela fondamentale riuscire a studiare e individuare gli attacchi informatici che minano la stabilità di questi nuovi tipi di sistemi distribuiti. In questa tesi proponiamo la realizzazione di modelli di simulazione di attacchi di Distributed Denial of Service e di Man In The Middle sul protocollo di comunicazione prevalentemente adottato per il controllo e il monitoraggio delle risorse energetiche distribuite: il Manufacturing Message Specification. Esploreremo le diverse fasi del processo di progettazione e sviluppo delle varie configurazioni costruite grazie al framework *OMNeT++* e alle librerie di supporto *INET* e *Simu5G*. Analizzando le misure e i log prodotti dalle simulazioni siamo in grado di stabilire in che modo è caratterizzato l'attacco, per poi riuscire in futuro a definire dei dataset utilizzabili per il miglioramento di sistemi d'Intrusion Detection.

Indice

1 Introduzione **3**

2 Contesto e scopo del progetto **5**
2.1 Sistemi d'Intrusion Detection 5
2.2 Distributed Energy Resources 10
2.3 Scenari d'attacco 14
2.3.1 Distributed Denial of Service (DDoS) 14
2.3.2 Man In The Middle (MITM) 15

3 La simulazione **17**
3.1 La struttura di un simulatore 17
3.2 La simulazione a eventi discreti 20
3.3 La generazione dei numeri casuali 21
3.4 La modellazione degli input 22
3.5 L'analisi statistica dei modelli stocastici 24

4 Gli strumenti utilizzati **27**
4.1 Il framework OMNeT++ 27
4.1.1 I componenti dei modelli 28
4.1.2 Un esempio pratico: Source-Sink 32
4.2 Il framework INET 42
4.2.1 I componenti di INET 42
4.2.2 Le TCPApplications 46
4.3 La libreria Simu5G 50
4.3.1 I componenti di Simu5G 50

5 I modelli **53**
5.1 La fase di progettazione 53
5.2 Il modello MMS 53
5.2.1 La definizione del messaggio 54
5.2.2 Il client MMS 56
5.2.3 Il server MMS 67
5.2.4 Il client malevolo (DDoS) 75
5.2.5 Il client malevolo (MITM) 76
5.2.6 Le configurazioni realizzate: Man In The Middle 108
5.2.7 Le configurazioni realizzate: Distributed Denial Of Service . . 131

6 Analisi dei risultati **138**
6.1 L'attacco Man In The Middle 138
6.2 L'attacco Distributed Denial of Service 146
6.3 Tempo di esecuzione 150

7 Conclusioni e sviluppi futuri **155**

Capitolo 1

Introduzione

Questo lavoro di tesi nasce da una collaborazione tra il Dipartimento di Scienze e Innovazione Tecnologica dell'Università del Piemonte Orientale e l'ente di Ricerca sul Sistema Energetico (RSE) di Milano. Si tratta di un progetto di ricerca focalizzato sullo studio dei problemi posti da diversi tipi di attacchi informatici su sistemi di controllo di distribuzione dell'energia. Il nostro interesse principale riguarda la cosiddetta **Smart Grid** (SG) ossia l'insieme d'impianti distribuiti di produzione di energia rinnovabile (ad esempio solare ed eolica) che comunicano attraverso la rete Internet e necessitano di un monitoraggio costante.
Un primo approccio all'analisi di questi problemi ha previsto lo sviluppo di un *testbed* presso la sede di RSE, che attraverso tecniche di emulazione ha permesso di studiare il compimento di alcuni tipi di attacchi all'interno di scenari predefiniti. Se da una parte questa strategia permette di ottenere un'elevata fedeltà di riproduzione dei protocolli e dell'infrastruttura scelta, dall'altra non offre grandi possibilità di modifica della rete sia in termini di dislocazione dei componenti che dal punto di vista del numero in cui vengono dispiegati. In questo contesto si collocano le tecniche di **simulazione**, che seppur adottando un livello di astrazione più elevato, garantiscono una maggiore flessibilità dal punto di vista strutturale e prestazionale. Definendo così diverse topologie e configurazioni siamo in grado di studiare l'evolversi degli attacchi d'interesse in scenari differenti. Nonostante queste strategie possano sembrare due mondi inconciliabili, in realtà rappresentano due approcci complementari che possono essere usati in combinazione per migliorare sistemi d'**Intrusion Detection** (ID).
Nel capitolo 2 esploreremo più nel dettaglio il contesto in cui nasce questo progetto partendo dal comprendere come possono essere classificati i diversi sistemi d'ID, per poi descrivere come questi possono essere integrati nel testbed di emulazione sviluppato da RSE. Ci concentreremo poi sull'esposizione delle peculiarità dei sistemi di **Distributed Energy Resources** (DER) descrivendo come questi comunicano con le *substations* (possiamo vedere uno schema in Figura 1.1) grazie al protocollo **Manufacturing Message Specification** (MMS) e definito dallo standard IEC 61850. Ne analizzeremo la struttura logica elencando i tipi di messaggi previsti al suo interno per poi passare all'esposizione degli attacchi di *Distributed Denial of Service* e *Man In The Middle*, oggetto del successivo studio basato su simulazione.

Nel capitolo 3 sarà illustrata la struttura di un generico simulatore, per poi focalizzarci sugli elementi cardine della simulazione a eventi discreti. In quest'ottica vedremo aspetti relativi alla generazione dei numeri pseudocasuali, alla scelta delle

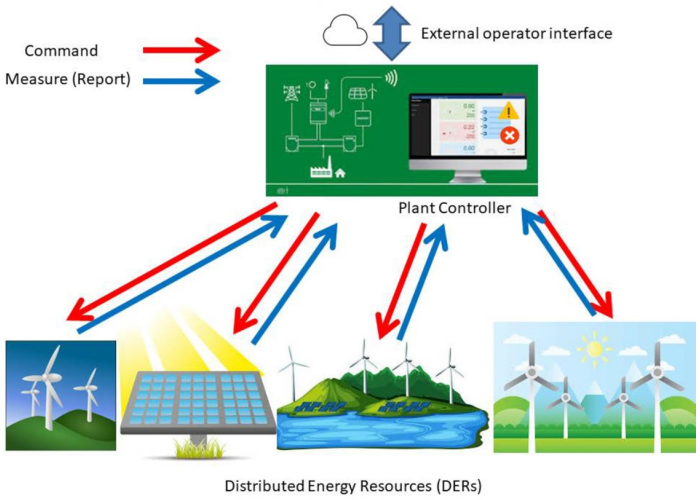


Figura 1.1: Controllo di Distributed Energy Resources

distribuzioni di input e all'analisi statistica dei modelli stocastici (sia in termini di stime che di presentazione grafica).
Passeremo poi nel capitolo 4 ad analizzare gli strumenti adottati tra cui il framework di simulazione ***OMNeT++***, fondamentale nella modellazione di reti basate sullo scambio di messaggi e di cui esporremo i concetti principali anche attraverso un esempio pratico. Nella parte seguente vedremo le librerie di supporto ***INET*** (per l'introduzione delle funzionalità e dei protocolli peculiari dallo stack TCP/IP) e ***Simu5G*** (per la simulazione delle connessioni e dell'infrastruttura 5G).
Nel capitolo 5 saranno descritti dettagliatamente i componenti dei modelli implementati partendo da client e server necessari al funzionamento normale del protocollo MMS. Successivamente si illustrerà come sono stati realizzati i client malevoli adottati negli attacchi di DDoS e MITM, andando a scomporli nelle loro componenti fondamentali e studiandone il funzionamento. Nelle sezioni finali analizzeremo le configurazioni implementate per studiare i rispettivi attacchi osservando i moduli, le connessioni e i parametri che le caratterizzano.
Per estrarre qualche informazione interessante nel capitolo 6 avremo modo di capire quali sono le misure e le statistiche che più evidenziano la presenza degli attacchi in ciascuno scenario e potremo anche valutare se e come le diverse tecnologie di comunicazione (Ethernet, Wi-Fi e 5G) intervengono nella caratterizzazione del modello.
Infine nel capitolo 7 concluderemo la nostra analisi discutendo degli sviluppi futuri del progetto e di come pensiamo possa essere esteso per fornire uno strumento utile a migliorare l'efficacia di sistemi d'ID.

Capitolo 2

Contesto e scopo del progetto

In questo capitolo esporrò gli elementi che caratterizzano il contesto in cui si colloca questo progetto di tesi e analizzerò più approfonditamente alcune delle tematiche che definiscono l'ambiente per cui questo lavoro è stato realizzato. Partendo dalla comprensione di cosa sono i sistemi di 'Intrusion Detection' spiegherò come questi siano necessari anche per proteggere i moderni apparati distribuiti di produzione dell'energia (DER). Proseguiremo successivamente presentando il protocollo MMS tramite il quale è possibile interagire con questi sistemi, andandone a studiarne le diverse componenti. Infine introdurremo i principali scenari di attacco presi in considerazione per lo studio.

2.1 Sistemi d'Intrusion Detection

Con il termine "Intrusion Detection System" (IDS) [7] definiamo tutti quei sistemi e algoritmi che hanno lo scopo di rilevare qualunque attività volta a intaccare la confidenzialità, l'integrità o la disponibilità di un certo sistema informativo, così da poter adottare le contromisure adeguate. Un IDS deve essere in grado di riconoscere traffico di rete e log d'utilizzo malevoli, i quali non possono essere identificati tramite l'uso di un comune firewall a causa della loro intrinseca interpretazione sequenziale.

In letteratura è possibile trovare una gran varietà di tecniche diverse applicate al problema dell'Intrusion Detection. Possiamo operare una prima grande distinzione tra i diversi modelli sulla base del principio su cui si fondano:

- **Signature-based intrusion detection systems**: All'interno di una base di conoscenza vengono inserite tutte le firme che identificano attacchi noti. Se viene rilevata una sequenza di pacchetti o comandi la cui firma corrisponde a una già presente nel database allora viene sollevato un avvertimento che notifica la presenza di un'intrusione. Nonostante questi sistemi siano stati i primi a essere inventati, essi presentano diverse criticità come la difficoltà nel rilevare gli attacchi zero-day (finché la loro firma non è inserita nel database) e la complessità nel comprendere su quali sequenze di pacchetti vada generata la firma.
- **Anomaly-based intrusion detection systems**: Comprende quell'insieme d'IDS formato da tutti quegli algoritmi che costruiscono un modello basato

sulla definizione di "normalità" del sistema informativo. Ciò implica che debba essere valida l'assunzione per cui il comportamento di un normale utente differisca significativamente dal comportamento dell'attaccante, il quale può così essere identificato come un'attività anomala. Questi sistemi operano secondo due fasi distinte, quella di "*training*" in cui si addestra il modello a riconoscere il traffico normale e quella di "*test*" in cui si verifica che l'algoritmo sia in grado di generalizzare a partire dai dati inizialmente osservati, di modo che possa così distinguere l'attività sospetta da quella normale. Il vantaggio principale di questo tipo di sistemi è che il modello non deve contenere al suo interno tutti i tipi di attacco possibili sotto forma di firme, ma solo ciò che consideriamo come attività regolare del sistema.

Nello studio dei diversi sistemi d'Intrusion Detection dobbiamo operare un'importante distinzione tra le diverse sorgenti dati utilizzabili per la rilevazione dell'intrusione. Un IDS di tipo **host-based** analizza i dati prodotti da una macchina host sotto forma di log del firewall, del sistema operativo e del database. Invece un IDS di tipo **network-based** si basa sul monitoraggio del traffico di rete che intercorre tra diverse macchine anche disposte in punti topologicamente distinti. Nonostante questo tipo di tecnica presenti alcune complicazioni nell'analisi di un traffico di rete elevato, offre la possibilità di proteggersi sia da attacchi provenienti dall'esterno della rete che da minacce provocate da *insider*.

Concentrandoci sui sistemi d'Intrusion Detection basati sul rilevamento delle anomalie possiamo ulteriormente differenziarli sulla base della tecnica adottata nel loro funzionamento, secondo la seguente tassonomia:

- **Statistical based**: Questa categoria d'IDS costruisce un modello statistico che rappresenta la condizione di normalità di un certo flusso di pacchetti. In pratica vengono tenuti in considerazione dei parametri come media, mediana e deviazione standard di variabili associate ai singoli pacchetti e sulla base di queste è possibile andare a calcolare la probabilità che un determinato pacchetto rappresenti un tentativo d'intrusione. Vengono così definite delle distribuzioni teoriche (es. Normali) che sono parametrizzate sulla base dei suddetti valori e tramite la loro funzione di densità di probabilità si va a determinare la probabilità di "normalità" di un pacchetto. Se è definita una distribuzione di probabilità separatamente per ogni variabile allora la strategia applicata si dice **monovariata**, mentre se viene definita un'unica distribuzione di probabilità che tenga in considerazione tutti i parametri contemporaneamente allora la tecnica si dice **multivariata**. Quest'ultima strategia presenta alcune complicazioni nell'essere applicata a dataset dall'elevata dimensionalità, ma generando un'unica distribuzione di probabilità multivariata permette di prendere in considerazione le dipendenze che sussistono tra variabili diverse. Infine esiste una tecnica che sfrutta le correlazioni che sussistono nelle sequenze di dati come le **serie temporali**, in cui il modello statistico generato restituisce la probabilità che una determinata osservazione occorra a quell'istante di tempo. Se la probabilità che ciò avvenga è eccessivamente bassa allora tale pacchetto può essere catalogato come un'anomalia.

- **Knowledge based**: Questo insieme di tecniche si basa sulla costruzione di una base di conoscenza che contenga le definizioni di tutti i profili di traffico

legittimi. Una tecnica d'implementazione si basa sull'uso delle **Macchine a Stati Finiti** (FSM) in cui il sistema è rappresentato sotto forma di stati, transizioni e attività che caratterizzano la normale operatività del sistema e ogni deviazione da quel flusso d'esecuzione viene segnalata come un tentativo d'intrusione. Se da un lato questo meccanismo evita di avere un elevato tasso di falsi positivi, dall'altro necessita di un aggiornamento continuo vista la costante evoluzione dell'attività degli utenti e tale operazione risulta onerosa in termini di tempo.

- **Machine learning**: L'ultima categoria è quella degli IDS basati su algoritmi di Machine Learning. In particolare riferendoci alle tecniche *supervised* includiamo tutti quei metodi che sono in grado di apprendere parametri e regole, necessari per il loro funzionamento, a partire dai dati stessi. La costruzione di un algoritmo di questa tipologia si suddivide in due fasi: *training* e *test*. In fase di training si sottopone all'algoritmo una serie di pacchetti o log a cui vengono associate delle etichette come "normale" o "intrusione" e attraverso un apposito meccanismo di apprendimento va a imparare le relazioni che sussistono tra le features e le rispettive etichette. In fase di test invece al modello addestrato vengono sottoposti dei sample mai usati in fase di addestramento e viene così verificata la sua capacità di generalizzare. Un esempio di algoritmo di Machine Learning supervisionato è quello dei **Decision Tree**, formato da diversi nodi decisionali sui quali vengono eseguiti dei test su diversi attributi. Per ogni ramo figlio di un nodo effettuiamo delle decisioni basate su valori possibili della feature associata a quel nodo. Alla base dell'albero sono presenti le foglie, ognuna delle quali rappresenta la classe di appartenenza del caso fornito in input. Seppur questa strategia presenti un'elevata *spiegablità* (è facile comprendere su che basi funzioni la classificazione) è piuttosto prona all'*overfitting*, infatti minime variazioni del training set causano grandi cambiamenti nel modello risultante. Un altro algoritmo facente parte di questa categoria sono le **Reti Neurali Artificiali** (ANN) che attraverso una serie di livelli formati da componenti fondamentali detti *percettroni* associano una predizione a un certo input. Tale meccanismo sfrutta una serie di parametri appresi in fase di addestramento mediante la tecnica della ***backpropagation*** (BP) in cui a ogni iterazione vengono modificati tali parametri della rete in modo da minimizzare l'errore in predizione. Questo permette di modellare funzioni non lineari anche molto complesse, pressoché impossibili da ricavare in maniera analitica. Nonostante questo approccio sia ampiamente utilizzato risulta spesso poco efficace nella rilevazione di attacchi poco frequenti e non inclusi nel dataset di training. Infine un ultimo esempio di questo tipo è rappresentato dagli **Algoritmi Genetici**, una tecnica euristica basata su principi evoluzionistici, in cui si fa evolvere per iterazioni successive una popolazione d'individui (soluzioni codificate tramite valori interi o sequenze binarie dette *geni*) in modo da selezionare e far riprodurre con probabilità maggiore gli esemplari che meglio risolvono il problema in esame. Nel caso dell'intrusion detection la popolazione iniziale è composta di regole casuali riguardanti il traffico di rete e codificate sotto forma di *geni*. Queste vengono fatte accoppiare ed evolvere in generazioni successive fino a ottenere soluzioni che bene individuano il traffico malevolo.

Negli ultimi anni si stanno diffondendo anche tecniche di **unsupervised learning** in cui si suppone di non possedere alcuna etichetta associata ai campioni nel dataset, ma vengono generati dei raggruppamenti sulla base di un concetto di similarità/distanza calcolabile direttamente sui dati. Ciò che emerge dall'utilizzo di questi algoritmi di apprendimento automatico e in particolare nelle loro varianti "profonde" (es. Deep Neural Networks) [14] è la necessità di possedere una grande quantità di dati in fase di addestramento. Purtroppo però se da una parte molti dei dataset che rappresentano attacchi di rete risultano essere privati per questioni di privacy e sicurezza, dall'altra quelli resi pubblici sono stati anonimizzati rendendoli così non sempre applicabili ai casi di studio.

Come evidenziato in [1] oltre alla modellazione e all'implementazione del modello, un'altra fase importante è quella della **validazione** e proprio con questo obiettivo è stato proposto il framework mostrato in Figura 2.1. Lo scopo di tale studio era quello di emulare l'attività di un attaccante su un'architettura di rete predefinita e registrare il traffico all'interno di un database, in modo che potesse poi essere inoltrato (tramite l'uso di un broker) a diversi modelli di detection, così da poterne confrontare le prestazioni.

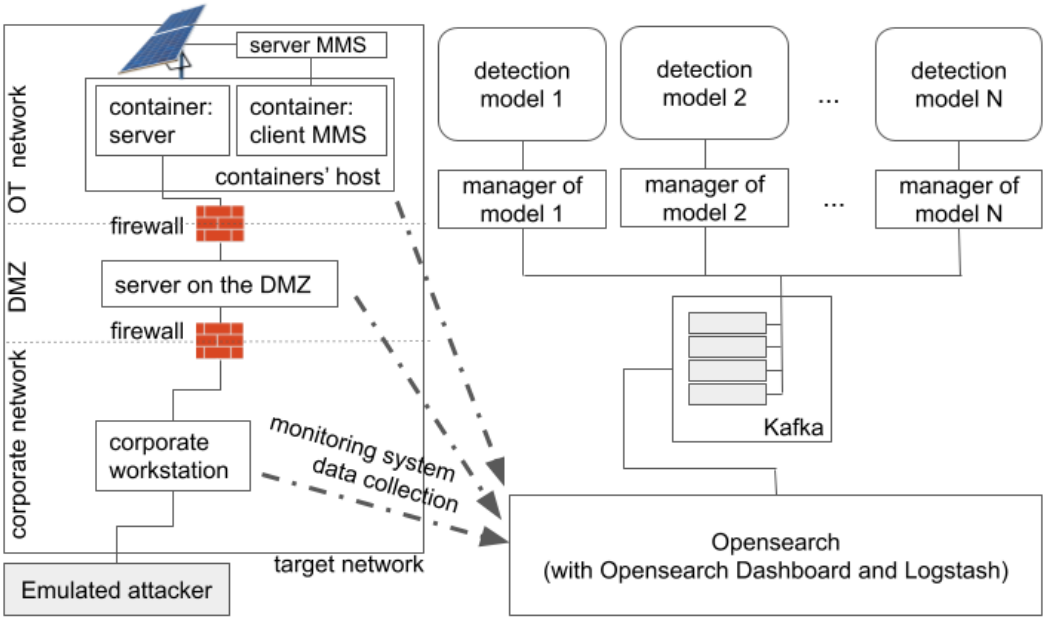


Figura 2.1: Framework per la validazione d'IDS in ambito energetico

I modelli inizialmente integrati in tale framework sono quelli delle Reti Bayesiane Dinamiche ma è necessario introdurre prima la loro variante statica per poterne comprendere al meglio le possibilità d'applicazione.
Facente parte della categoria dei modelli grafico-probabilistici, quello delle **Reti Bayesiane** (BN) è un formalismo molto potente, che permette di modellare il comportamento incerto attraverso una struttura ben definita. Una Rete Bayesiana è una coppia $\langle G, P \rangle$ in cui G è un **Grafo Diretto Aciclico** (DAG) e P è una distribuzione di probabilità sui nodi di G. Il DAG è formato a sua volta da $\langle V, E \rangle$, dove V è l'insieme dei nodi definito come un insieme di variabili casuali $\{X_1, ..., X_n\}$ ed E è l'insieme degli archi che rappresenta le dipendenze tali per cui se $e \in E$ ed e

va dal nodo X al nodo Y allora Y dipende direttamente da X. Ogni variabile è influenzata esclusivamente dai nodi genitore nel grafo, questo implica che per calcolare la **probabilità congiunta** di tutte le variabili è sufficiente ricavare la probabilità condizionale di ciascuna variabile rispetto ai nodi genitore nel DAG come mostrato nell'equazione 2.1.

$$P[X_1, ..., X_n] = \prod_{i=1}^{n} P(X_i|Parent(X_i)) \tag{2.1}$$

Nel caso in cui le variabili casuali siano discrete possiamo descrivere la distribuzione di probabilità di ogni singolo nodo attraverso una **Conditional Probability Table** (CPT) in cui è presente una colonna per ogni possibile combinazione dei valori delle variabili genitore. Se osservassimo i valori di alcune variabili, questi potrebbero essere inseriti nella rete sotto forma di **evidenza** permettendoci così di aggiornare le probabilità marginali nella rete sulla base di ciò che è stato specificato. Tutti i nodi nella rete che non fanno parte dell'evidenza sono definiti come **nascosti**. In generale si dice che una Rete Bayesiana è un **modello isotropico** nel senso che può essere usata sia per svolgere **ragionamento predittivo** (dalle cause alle conseguenze seguendo le direzioni dei vari archi) che per compiere **ragionamento diagnostico** (dagli effetti alle cause percorrendo il DAG in direzione opposta rispetto a quella degli archi). Un esempio di applicazione di questo specifico formalismo è visibile in [11], dove a partire dalla rappresentazione di diversi scenari d'attacco attraverso gli **Attack Trees** (AT) viene definire la struttura delle corrispettive Reti Bayesiane (nodi e archi), le cui probabilità vengono aggiornate sulla base di dati ottenibili tramite esperti di dominio o consultando basi di conoscenza preesistenti.
Se da una parte combinando il potere modellante degli Attack Trees con quello inferenziale delle Reti Bayesiane sia una scelta piuttosto efficace, dall'altra il fatto di usare quest'ultimo formalismo nella sua variante statica pone alcuni limiti nella modellazione degli attacchi. Per comprendere al meglio come si sviluppi un'intrusione è fondamentale avere la possibilità studiare la sua evoluzione nel tempo (anche se solo tramite una approssimazione discreta). Per poter fare ciò bisogna estendere le Reti Bayesiane per poter esprimere le relazioni di causalità tra i diversi nodi in istanti di tempo consecutivi. Proprio per questo, come mostrato in [1], quello delle **Reti Bayesiane Dinamiche** (DBS) è un formalismo che si applica particolarmente bene alla realizzazione di sistemi d'intrusion detection. Considerando un insieme di variabili $X_1, ..., X_n$ la rispettiva DBN non è altro che una replica degli N nodi su ciascuna fetta temporale su t e $t - \Delta$ (dove Δ è l'intervallo usato per discretizzare la dimensione temporale) e l'aggiunta di una serie di archi che collegano nodi d'istanti di tempo successivi. Sia quindi X_i^t la copia della variabile X_i al tempo t, definiamo il modello di transizione come la distribuzione $P[X_i^t|X_i^{t-\Delta}, Y^{t-\Delta}, Y^t]$ dove $Y^{t-\Delta}$ è un qualunque insieme di variabili al tempo $t - \Delta$ diverso da X_i e Y^t è un qualunque insieme di variabili al tempo t diverse da X_i. Un arco che collega $X_i^{t-\Delta}$ al tempo $t - \Delta$ alla variabile X_i^t è chiamata *arco temporale* e se $i = j$ allora l'arco collega due istanze della stessa variabile. Come visibile in Figura 2.2, questa strutturazione ci permette di far dipendere il valore di un nodo in un certo istante di tempo, oltre che dal valore dei suoi genitori, anche dal valore che possedeva all'istante di tempo precedente (le probabilità condizionate associate al nodo B_t saranno determinate sulla base di quelle contenute in A_t e in B_{t-1}).

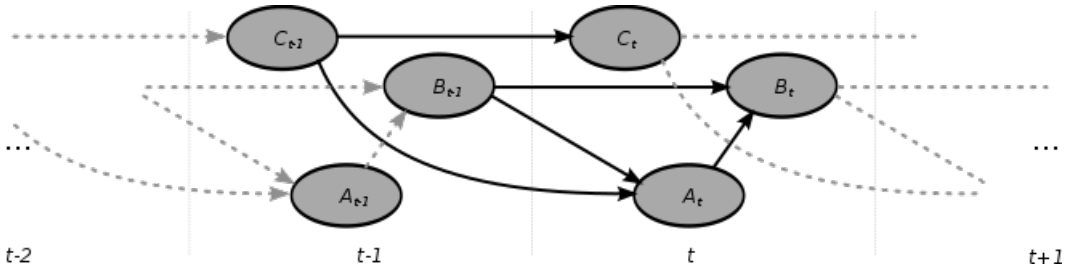


Figura 2.2: Esempio di Rete Bayesiana Dinamica

Nelle Reti Bayesiane Dinamiche possiamo svolgere delle inferenze in modo più raffinato rispetto a quelle disponibili per le BN. In questo caso l'*evidenza* viene accompagnata da un timestamp e si vuole determinare la probabilità di un insieme di variabili in un certo istante di tempo. I compiti che possono essere svolti con questa tecnica sono diversi e in parte risultano essere delle estensioni delle strategie già citate precedentemente:

- **Predizione**: Sulla base delle analitiche fornite possiamo individuare quali saranno le tecniche più probabili che l'attaccante sfrutterà in futuro, così da poter intraprendere le adeguate azioni difensive. Ciò implica calcolare le probabilità a posteriori al tempo t di un insieme d'ipotesi Q data la serie temporale di osservazioni $e_{t_1}, ..., e_{t_k}$ (insieme di variabili istanziate per ciascuna fetta temporale) dal tempo t_1 al tempo t_k con $t_1 < ... < t_k < t$.

- **Filtraggio**: Rappresenta un caso speciale di ragionamento predittivo in cui l'ultimo istante temporale per cui è stata fornita dell'evidenza e quello in cui sono eseguite le query d'inferenza coincidono ($t = t_k$). Questa strategia è particolarmente utile nell'eseguire intrusion detection in tempo reale.

- **Smoothing**: Permette di svolgere ragionamento diagnostico andando a determinare le possibili cause di problemi di sicurezza. Si vanno quindi a calcolare le probabilità di uno stato passato date le evidenze appartenenti a uno stato futuro. In altre parole questo significa calcolare le probabilità al tempo t di un insieme d'ipotesi Q data la serie temporale di osservazioni $e_{t_1}, ..., e_{t_k}$ (insieme di variabili istanziate per ciascuna fetta temporale) dal tempo t_1 al tempo t_k con $t < t_1 < ... < t_k$.

2.2 Distributed Energy Resources

Con il termine **Distributed Energy Resources** ci riferiamo oltre che a tutti quei sistemi di generazione dell'energia come celle a combustibile, micro-turbine e fotovoltaico, anche alle diverse tecnologie di stoccaggio dell'energia prodotta come batterie, superconducting magnetic energy storage (SMES) o accumulatori a volano. In genere questi sistemi di generazione dell'energia sono collegati a una rete di distribuzione a medio o basso voltaggio, da cui l'energia può essere veicolata all'utente finale. Negli apparati moderni tale rete viene anche definita come **Micro Grid** (MG) e può essere collegata alla rete di distribuzione esterna tramite un *Point of Common Coupling* (PCC). Inoltre uno switch permette d'isolare questa rete da

quella esterna quando si verificano guasti o problemi di altra natura. Il flusso di energia può avvenire in maniera bidirezionale e tale direzione dipende dal carico locale, dalla fornitura esterna o dal prezzo dell'energia. Un sistema di monitoraggio intelligente ha lo scopo di gestire la comunicazione con la rete esterna, programmare la produzione di energia, lo stoccaggio e la richiesta dall'esterno. Ad esempio in un impianto fotovoltaico può decidere di dirottare l'energia generata nel sistema di accumulo locale durante il giorno, per poi immetterla nella rete esterna se questa è prodotta in esubero.
Negli ultimi anni si è reso sempre più necessario lo sviluppo di quelle che vengono definite come **Smart Grid** (SG) ossia sistemi intelligenti di distribuzione dell'energia in grado di reagire proattivamente in base alla domanda, alla disponibilità, e ai guasti, specialmente se accoppiati a sistemi di fornitura di energia rinnovabile. Per adempiere a questo scopo è necessaria l'adozione di una serie di paradigmi e tecniche che attingono da diversi ambiti dell'informatica come l'analisi dei Big-Data, l'utilizzo di tecniche d'Intelligenza Artificiale e di sistemi classificati come Internet Of Things (IoT). La comunicazione tra i diversi componenti di questa SG avviene in maniera bidirezionale e si basa sull'uso di tecnologie di trasmissione dati come le reti cablate, il Wi-Fi o il 5G. Rendere sicuri questi scambi di dati rappresenta una delle principali sfide nell'implementazione di tali infrastrutture.
Un componente fondamentale per il corretto funzionamento dei DER è il **Manufacturing Message Specification** [1] (MMS), il quale è un protocollo per lo scambio di dati e comandi in tempo reale riconosciuto internazionalmente (ISO 9506). Ufficializzato nel 1986, MMS si è subito imposto nell'industria meccanica e in quella elettrica rispettivamente per la gestione di CNCs/PLCs (macchine a controllo numerico) e degli Energy Management Systems (EMS utilizzati all'epoca nell'ambito della generazione dell'energia centralizzata). Questa versatilità è anche data dal fatto che MMS può funzionare al di sopra di diversi protocolli come Ethernet, Token Bus, TCP/IP ecc...
Sono principalmente tre i benefici portati dall'adozione di MMS:

- **Interoperabilità**: È la capacità di due applicazioni di scambiarsi dati e messaggi di controllo senza bisogno che l'utente definisca l'ambiente di comunicazione.
- **Indipendenza**: Permette di ottenere il requisito d'interoperabilità indipendentemente dallo *sviluppatore dell'applicazione* (grazie al fatto che MMS sia stato definito da un ente internazionale con la collaborazione di diverse aziende, non risulta specifico per l'uso di una sola applicazione), dalla *connettività di rete* (MMS diventa l'interfaccia di rete per le applicazioni nascondendo loro tutti gli aspetti di trasmissione dati da un nodo all'altro) e dalle *funzionalità implementate* (MMS fornisce un ambiente di comunicazione comune che permette ad applicazioni che nascono in contesti diversi di avere un protocollo standard per la comunicazione con certi dispositivi).
- **Accessibilità**: Applicazioni diverse possono accedere alle informazioni necessarie per il loro funzionamento.

Mentre molti schemi di comunicazione forniscono solo un meccanismo per trasmettere una sequenza di *bytes* (messaggi) attraverso la rete, MMS provvede anche

[1] Fare riferimento a [12] per la specifica completa

alla definizione della struttura e del significato dei messaggi permettendo così di comunicare a due applicazioni sviluppate indipendentemente.

Il protocollo MMS si fonda su un modello di comunicazione chiamato **Virtual Manufacturing Device** (VMD) e specifica come i dispositivi MMS (server) si comportano dal punto di vista di un client esterno (schema visibile in Figura 2.3). In generale nel modello VMD sono definiti i seguenti elementi:

- **Oggetti**: Rappresentano il componente fondamentale di MMS e possono essere trovati in molti dispositivi e applicazioni che richiedono di poter comunicare in tempo reale. Sono ad esempio:
  - *Variabili*: Contenute nel server, possono essere accedute tramite nome o tramite indirizzo. In MMS è anche possibile specificarne il tipo tramite un oggetto separato, infatti queste variabili possono essere sia semplici (interi, booleani, stringhe) che complesse (array e strutture).
  - *Program Control Objects*: Consistono in programmi eseguibili dalle applicazioni.
  - *Event Objects*: Usati per la gestione di eventi.
  - *Semaphore Objects*: Utili per il controllo di risorse condivise.
  - *Journal Objects*: Un record temporale di eventi e variabili.
  - *Files*: Sistema di memorizzazione dei file.
- **Servizi**: Regolano l'accesso agli oggetti in lettura e scrittura.
- **Comportamento**: MMS definisce il comportamento che un dispositivo deve avere quando esegue i suddetti servizi.

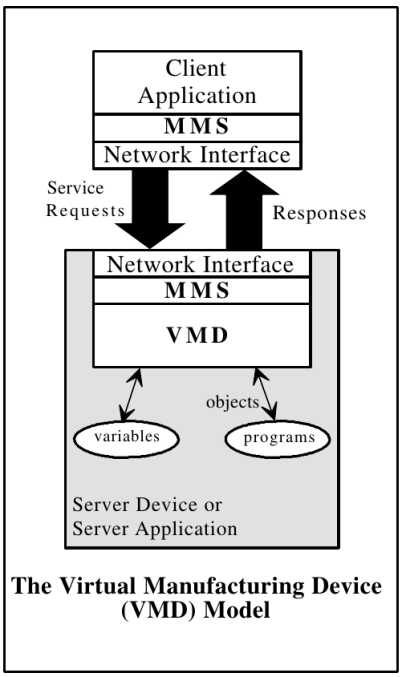


Figura 2.3: Virtual Manufacturing Device (VMD) Model

Nonostante la specifica completa di MMS comprenda maggiori dettagli, molti di questi risultano superflui per gli scopi della nostra simulazione. L'aspetto principale che ci interessa modellare del VMD model è la relazione **client/server** che sussiste tra le applicazioni di rete e/o i dispositivi (un esempio è visibile in Figura 2.4). Un server è un'entità che contiene una serie di oggetti (es. variabili) specificati secondo il VMD, mentre un client è un'applicazione di rete che invia richieste MMS a un server. Qualunque dispositivo MMS che offra dei servizi secondo il protocollo MMS di un server deve seguire il modello VMD per tutti gli aspetti di rete in esso visibili. I client MMS invece devono solamente attenersi alle direttive per la strutturazione dei messaggi e rispettare la sequenza d'invio dettata dal protocollo.

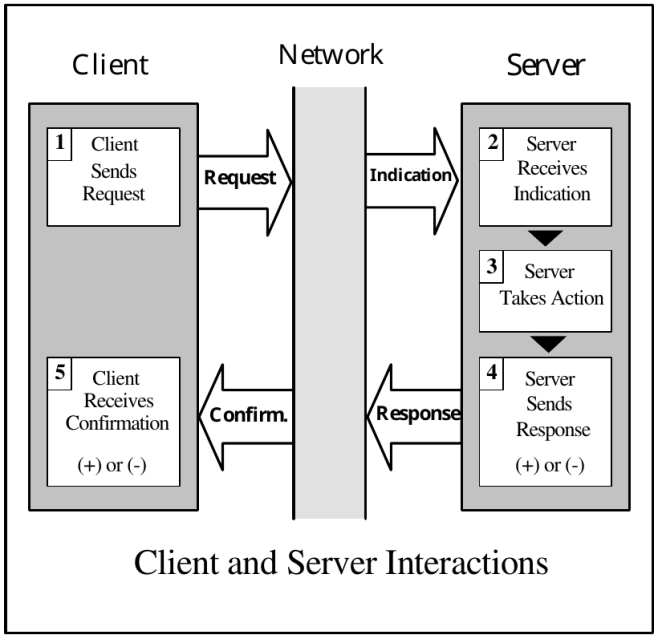


Figura 2.4: Comunicazione MMS Client/Server

Volendo implementare un sottoinsieme ridotto delle funzionalità definite da MMS abbiamo scelto le seguenti:

- **Lettura di variabili**: Il client invia al server MMS una richiesta di lettura di una variabile (tramite nome o indirizzo) e riceve in risposta il valore contenuto in tale oggetto MMS. Ad esempio un'applicazione potrebbe voler conoscere il valore istantaneo di una certa misura contenuta sul server.

- **Invio di comandi**: Il client invia al server una richiesta di esecuzione di un comando/programma e il server MMS risponde con il risultato dell'operazione.

- **Ricezione periodica di valori**: Si tratta dell'invio periodico da parte del server MMS di valori/misure associate a oggetti per cui il client si era inizialmente sottoscritto. Questa azione è simile all'operazione di lettura di variabile ma senza che il client inizializzi ogni volta la comunicazione con un messaggio di richiesta. Questa funzionalità (definita dal protocollo MMS come *InformationReport*) è stata introdotta nel protocollo per eliminare la necessità di un meccanismo di polling da parte del client per l'aggiornamento delle misure.

Come vedremo ora in sezione 2.3 possono essere sferrati diversi tipi di attacco volti a minare i requisiti fondamentali di sicurezza del protocollo MMS: attacchi che alterano il normale flusso di comunicazione tra client e server, ed altri che mirano a bloccare il server MMS generando una grande quantità di richieste fasulle.

2.3 Scenari d'attacco

Fino a ora ci siamo concentrati sullo studio del normale funzionamento di una comunicazione con protocollo MMS, ma spesso può accadere che un malintenzionato tenti di compromettere la sicurezza di questi scambi per fini illeciti. I requisiti principali che vorremmo garantire sono tre:

- *Riservatezza*: Anche definito come confidenzialità, è la capacità di escludere che l'informazione sia fruita da persone o risorse sprovviste di esplicita autorizzazione.
- *Integrità*: È la condizione per cui le informazioni rimangono inalterate e coerenti, salvo nel caso in cui debbano essere apportate modifiche autorizzate.
- *Disponibilità*: È la situazione in cui le informazioni sono prontamente raggiungibili, nel momento in cui sono richieste.

Nelle sezioni 2.3.1 e 2.3.2 che seguono verranno presentati ad alto livello gli attacchi presi in considerazione. I dettagli della loro implementazione all'interno di modelli sono riportati più avanti nel capitolo 5.

2.3.1 Distributed Denial of Service (DDoS)

Gli attacchi di **Distributed Denial of Service** (DDoS) hanno lo scopo di minare la disponibilità del server MMS, andando a sovraccaricarlo con un volume elevato di richieste malevole, in modo che i messaggi provenienti dai client reali non possano essere gestiti in tempi utili. L'aggettivo "*Distributed*" si riferisce al fatto che l'attaccante non si espone direttamente durante il compimento dell'attacco, ma sfrutta un *Command and Control Server* (C&C) che controlla l'offensiva istruendo una serie di host compromessi (che nel loro insieme sono definiti come *botnet*) sull'invio di richieste al server di destinazione. Uno schema esemplificativo contenente i componenti di questo attacco è mostrato in Figura 2.5.

Questo attacco è particolarmente insidioso per due motivi principali:

- **Facilità di attuazione**: Per il compimento di questo attacco non sono richieste competenze informatiche avanzate e parte delle procedure comunemente adottate fa affidamento su programmi o servizi semiautomatici che abbassano il livello di abilità richiesto per la sua corretta esecuzione.
- **Difficoltà di rilevazione**: A causa del traffico distribuito su diversi host malevoli, la distinzione tra richieste reali e fittizie è resa più complessa. L'utilizzo di un firewall per il blocco di singoli IP potrebbe non essere sufficiente ad arginare il traffico proveniente da botnet che cambiano in continuazione.

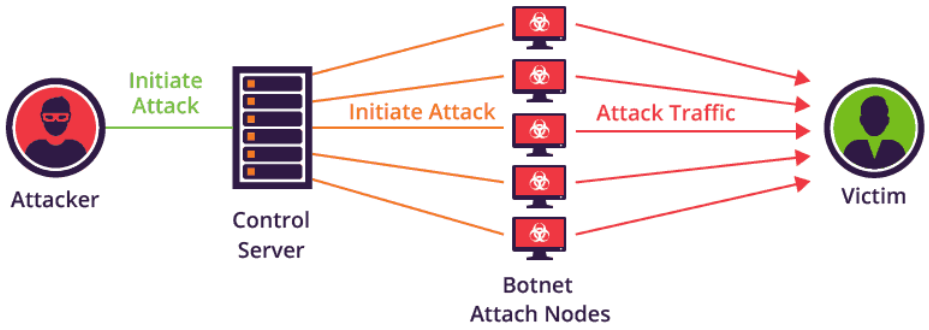


Figura 2.5: Schema di un attacco di *DDoS*

All'interno dei nostri modelli supponiamo che l'attaccante sia già riuscito a compromettere e impartire alla botnet i comandi di avvio dell'attacco, quindi fin da subito gli host malevoli (in seguito alla connessione e alla sottoscrizione all'*Information Report* sul server MMS) procedono a inviare richieste di lettura di variabili (false) a un tasso ben più alto di quello adottato dai client reali.

2.3.2 Man In The Middle (MITM)

L'attacco **Man In The Middle** (MITM) riguarda quella tipologia di compromissioni in cui l'attaccante si frappone fra *client* e *server* (senza che le parti coinvolte ne siano a conoscenza) e si avvale della facoltà di monitorare il traffico (*sniffing*) e bloccare o falsificare i messaggi scambiati in una o entrambe le direzioni. Ciò che normalmente avviene è che l'attaccante punta ad effettuare uno "split" della connessione TCP (client-attaccante e attaccante-server), di modo che sia il mittente che il destinatario siano convinti di stare comunicando con la fonte reale dei messaggi (questa operazione richiede una serie di passaggi complessi se il canale da "splittare" è cifrato).

Per esprimere formalmente i passi seguiti dall'attaccante per l'esecuzione del MITM, usiamo il formalismo dei **grafi d'attacco** in cui ciascun nodo rappresenta uno stato in cui l'attaccante si può trovare, mentre un arco che collega due nodi rappresenta l'azione che comporta il passaggio da uno stato al successivo. L'attacco in esame segue gli step mostrati in Figura 2.6, i quali fanno riferimento ciascuno a una specifica tattica della classificazione MITRE ATT&CK [2] (tra parentesi è indicato il nome della tecnica con i dettagli relativi all'implementazione). Alcune tattiche sono prese dalla matrice Enterprise ATT&CK (ENT) [3], mentre altre sono prese da quella ICS [4]. La prima è uno strumento più generico usato per classificare attacchi informaci a sistemi IT tradizionali (in tutte le loro fasi), mentre la seconda è progettata specificatamente per organizzazioni che utilizzano sistemi di controllo industriale (come quelli usati in impianti di controllo e infrastrutture critiche). In particolare nel nostro caso in seguito alla compromissione della chiave privata del client e all'installazione della Certification Authority dell'attaccante su di esso, l'entità malevola attua lo split della connessione MMS su TLS e modificando le misure inviate nei report o alterando i comandi inviati dal client, porta il server MMS in uno stato instabile. Questo tipo di azione malevola può quindi essere portata a compimento su connessioni sicure come quelle basate su *TLS*, ma a causa delle

limitazioni imposte da *INET* [2] questo protocollo non è stato implementato esplicitamente nella simulazione. Nei nostri modelli ci siamo concentrati sulla realizzazione dei passi d'attacco che seguono la compromissione della connessione, in cui l'entità malevola sta già attuando le operazioni di spoofing e ha già concluso con successo l'azione d'intercettazione della comunicazione client-server, dopo aver compromesso la connessione. Se necessario i ritardi normalmente introdotti da TLS possono essere aggiunti senza la necessità di una sua modellazione completa in base alle rilevazioni eseguite in [13].
L'attacco MITM punta a minare tutti i requisiti di sicurezza elencati all'inizio di questa sezione, sia andando a impartire al server MMS comandi fittizi che il client non ha mai inviato, che trasmettendo pacchetti con misure alterate, di modo che il client abbia una visione alterata del sistema e si trovi quindi indotto a reagire con azioni dannose per il sistema energetico.

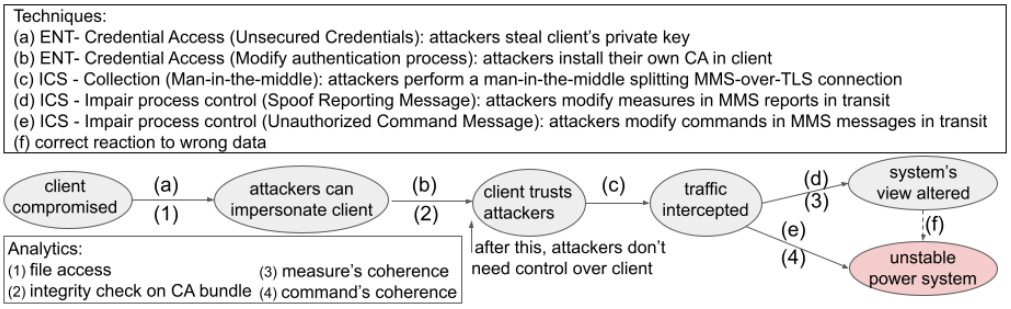


Figura 2.6: Grafo d'attacco MITM a protocollo *MMS-over-TLS*

[2] Vedere la sezione 4.2 per una spiegazione più dettagliata sul framework

Capitolo 3

La simulazione

La simulazione computerizzata è una tecnica che permette lo studio della realtà e del suo comportamento dinamico tramite la realizzazione di una sua rappresentazione semplificata. Una volta modellati gli aspetti fondamentali del sistema, incluse le regole che ne definiscono il funzionamento, il simulatore viene avviato e il suo comportamento viene tracciato attraverso delle variabili che possono essere analizzate a fine simulazione. La simulazione permette di studiare le interazioni che avvengono internamente a un sistema complesso e a ciascuno dei suoi componenti. L'adozione di diversi parametri di input e la conseguente generazione di output differenti ci fa comprendere quali siano le variabili che maggiormente influenzano i risultati ottenuti. Il poter simulare il sistema sotto diverse configurazioni (es. architetture di rete), garantisce una libertà d'azione difficilmente ottenibile tramite altri strumenti di analisi. In questo capitolo definiremo per prima cosa la struttura di un simulatore analizzandone tutte le componenti, discuteremo delle differenze tra *modelli deterministici* e *stocastici* ed esploreremo le peculiarità che caratterizzano un *simulatore a eventi discreti*. Ci concentreremo poi sui meccanismi che regolano la generazione di numeri pseudocasuali, sulla modellazione degli input e infine sull'analisi statistica delle variabili ottenute dalle varie *run* di simulazione.

3.1 La struttura di un simulatore

Un sistema di simulazione è solitamente costituito da un certo numero di componenti o entità descritte da una serie di attributi. Il primo passo da seguire nella costruzione di un simulatore è quello di delimitare la realtà che vogliamo studiare, separandola da tutto ciò con cui essa interagisce ma che ne è all'esterno (l'ambiente). In questo modo tutto ciò che includiamo nel nostro simulatore entrerà a far parte del suo *modello*. Di un certo sistema possono esistere diversi modelli che lo rappresentano e le tecniche di risoluzione/simulazione su cui ci possiamo basare sono diverse:

- **Metodi matematici**: Si possono ottenere soluzioni espresse in formule che mettono in relazione i parametri del sistema con le grandezze da misurare. Questi metodi sono molto precisi e relativamente poco costosi, ma spesso per essere applicati necessitano d'ipotesi esemplificative non sempre accettabili.
- **Metodi di simulazione**: Si studia l'evoluzione del modello nel tempo mimando come il sistema funziona nella realtà. Questa strategia è più flessibile della

precedente e non richiede ipotesi esemplificative per poter essere applicata, ma il costo d'esecuzione cresce all'aumentare della complessità del modello.

In questa sezione ci concentreremo sui modelli che sfruttano metodi di simulazione.

In generale un **modello** è un insieme di assunzioni espresse attraverso relazioni simboliche, matematiche o logiche, le quali sussistono tra gli oggetti d'interesse del nostro sistema. I componenti di un modello sono i seguenti (alcuni esempi sono visibili in Figura 3.1):

- **Stato del sistema**: Una collezione di variabili contenenti tutte le informazioni necessarie per descrivere lo stato del sistema in qualunque istante di tempo. Lo stato del sistema può cambiare in modo *continuo* o in modo *discreto*
- **Entità**: Qualunque oggetto o componente del sistema che necessita di una rappresentazione esplicita (es. un server, un cliente o una macchina).
- **Attributi**: Le proprietà di una di una certa entità (es. la priorità di un cliente in attesa).
- **Eventi**: Un avvenimento istantaneo che cambia lo stato del sistema (es. l'arrivo di un nuovo cliente).
- **Attività**: Una durata di tempo dalla lunghezza specificata (es. un tempo di servizio o d'inter-arrivo definibile attraverso delle distribuzioni statistiche), il cui momento d'inizio è conosciuto.
- **Ritardi**: Una durata di tempo dalla lunghezza indefinita, la quale non è conosciuta finché non si verifica un evento che ne determina la conclusione (es. il tempo di attesa di un cliente in coda dipende dai tempi di servizio dei clienti che lo precedono).

Abbiamo citato alcuni componenti che meritano di essere approfonditi ulteriormente. Le **attività** sono associate al loro evento di fine siccome gli eventi che sanciscono la fine di ciascuna di esse dipendono esclusivamente dal loro tempo d'inizio e dalla durata (che è nota nel momento in cui l'attività inizia). I **ritardi** invece rappresentano il tempo di attesa tra due attività consecutive e sono definiti in base al verificarsi di determinate condizioni del sistema (la loro durata non è conosciuta quando iniziano). Quelli che principalmente ci interessano nelle nostre simulazioni sono gli eventi di fine attività (incondizionati), in quanto completamente determinabili quando l'attività a cui sono associati inizia.
In fase di creazione del modello è necessario stabilire con che livello di dettaglio si vuole rappresentare la realtà d'interesse. La rappresentazione del nostro simulatore deve possedere un livello di dettaglio tale da poter tracciare correttamente le statistiche che ci interessa studiare, senza però eccedere nell'introdurre elementi inessenziali che non sono utili agli scopi delle nostre analisi. Questo perché un modello più astratto è sicuramente più semplice da produrre e più efficiente da eseguire, ma potrebbe trascurare dettagli importanti in grado d'influenzare i risultati delle nostre simulazioni. Al contrario un modello più dettagliato potrebbe rappresentare completamente tutti gli aspetti della realtà studiata (anche quelli per noi superflui) ma

Sistema	Entità	Attributi	Attività	Eventi	Variabili di stato
Banca	Clienti, operatori	Operazioni che i clienti possono richiedere ad un operatore	Effettuare una operazione	Arrivo in banca, Partenza dalla banca	Num. di clienti nella banca, Num. di operatori occupati
Sistema di telecomunicazioni	Messaggi, linee di collegamento, router	Velocità di trasmissione, dimensione buffer	Trasmissione del msg su una linea di collegamento, elaborazione msg in base al protocollo	Invio msg, arrivo msg a destinazione (finale o intermedia)	Numero di msg in attesa di trasmissione in ciascun punto della rete
Sistema di produzione	Macchine, sistemi di trasporto pezzi	Velocità di lavorazione, frequenza dei guasti, …	Effettuare la lavorazione di un pezzo, trasportare un pezzo ad una macchina. riparare un guasto	Guasto di una macchina, fine lavorazione di un pezzo	Numero di pezzi in attesa di lavorazione, stato delle macchine: occupate, libere, guaste

Figura 3.1: Esempi di sistemi con i relativi componenti

risulterebbe alla fine troppo esoso in termini di tempo d'esecuzione e richiederebbe di conoscere parametri non sempre disponibili.

Una grande distinzione che possiamo fare tra i diversi tipi di modelli esistenti è la seguente:

- **Modelli deterministici**: L'evoluzione del sistema modellato è completamente determinata dato lo stato iniziale. Per ogni stato di partenza le regole operazionali definiscono un solo stato di arrivo e un unico percorso nello spazio degli stati.
- **Modelli stocastici**: Dato uno stato iniziale possono esistere molteplici stati finali e molteplici percorsi nello spazio degli stati, a causa di alcune componenti di casualità introdotte nel modello.

Questa caratteristica dei *modelli stocastici* può essere applicata sia per generare i *tempi di fine attività* (es. utilizzo una variabile aleatoria per decidere il tempo impiegato da un servitore a gestire un nuovo cliente) che per definire *regole operazionali* non esprimibili in maniera deterministica. Nel primo caso stiamo introducendo una casualità sulla base di una distribuzione precedentemente calcolata che descrive una caratteristica del sistema, mentre nel secondo la scelta di adottare una strategia aleatoria può essere portata dal fatto che modellare una versione deterministica risulti troppo complesso. In questo modo anche le statistiche generate a fine simulazione non potranno più essere valutate come valori puntuali ma saranno delle variabili aleatorie vere e proprie, quindi sarà necessario eseguire più volte la simulazione per raccogliere un campione significativo di tali misure a partire dal quale cercare di stimare i parametri caratteristici della loro distribuzione di probabilità (vedremo meglio come fare in sezione 3.5).

3.2 La simulazione a eventi discreti

Per le nostre simulazioni ci interessano in particolare quelli che sono definiti come *modelli dinamici* ossia le rappresentazioni per cui viene tracciato il comportamento nel tempo. A questo punto resta da valutare come varia lo stato del sistema durante la simulazione e sulla base di questo criterio possiamo operare la seguente distinzione (un esempio è visibile in Figura 3.2):

- **Modelli continui**: Quelli in cui le variabili di stato cambiano in modo continuo (es. la portata del flusso d'acqua di un canale).
- **Modelli discreti**: Quelli in cui le variabili di stato cambiano solo in determinati insiemi di punti di tempo discreti (es. il numero di clienti in fila cambia solo quando un cliente arriva o se ne va). Questi sono quelli che ci interessano particolarmente.

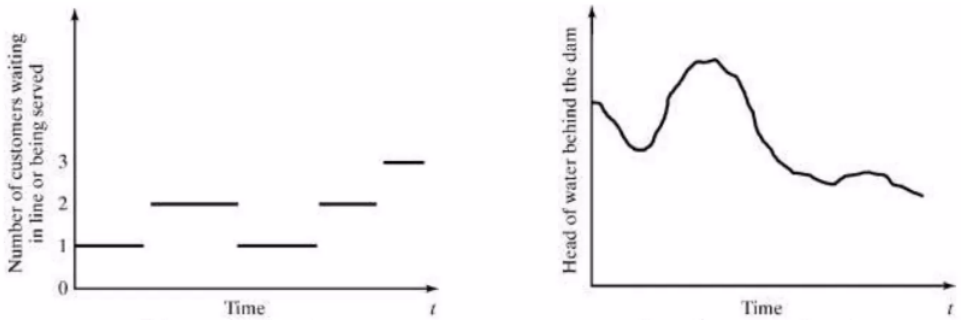


Figura 3.2: Modelli discreti e continui

Rispetto ai componenti definiti precedentemente dobbiamo introdurne un altro per permetterci di simulare questi sistemi a eventi discreti. Il componente caratterizzante di un simulatore a eventi discreti è la **Future Event List** (*FEL*) ossia una lista di *eventi* futuri in ordine cronologico, che grazie all'uso di un **clock** (tempo di simulazione) ci permette di avanzare in maniera incrementale all'interno dello spazio degli stati, memorizzando esclusivamente lo stato all'istante di tempo corrente. Bisogna solo preoccuparsi di aggiornare correttamente gli accumulatori necessari per il calcolo delle misure a fine simulazione. I simulatori a eventi discreti funzionano solitamente secondo un algoritmo chiamato **Next Event Time Advance** (NETA). In pseudocodice possiamo definirlo come segue:

1. *Inizializzazione* dello stato iniziale e inserimento nella *FEL* di tutti gli eventi futuri prevedibili (eventi di fine delle attività in corso).
2. Fino a quando ci sono eventi nella *FEL* e la simulazione non è terminata (dopo aver raggiunto il tempo massimo di simulazione o dopo che si è verificata una condizione prestabilita).
   (a) Si estrae il prossimo evento dalla *FEL* e si aggiornano *accumulatori* e *stato* in base alla situazione corrente e all'evento scattato.
   (b) Si imposta il *clock* al tempo di simulazione dell'evento estratto.

(c) Vengono inserite nella *FEL* tutte le *event notices* relative alle attività iniziate nello stato corrente e vengono rimosse quelle non più valide.

3. Calcolo delle misure al termine della simulazione sulla base degli accumulatori tracciati.

È possibile realizzare un simulatore a eventi discreti implementandolo direttamente in un qualunque linguaggio di programmazione, ma in tal caso saremmo costretti a gestire esplicitamente tutti gli aspetti di progressione della simulazione (es. avanzamento del clock, strutturazione della *FEL*, gestione delle event notice, gestione delle misure ecc...). Ciò che conviene fare invece è sfruttare dei framework che offrano già questo "scheletro" del simulatore e richiedano di definire esclusivamente le componenti base dei modelli, le loro regole operazionali e le statistiche che si vogliono ottenere al termine della simulazione. Vedremo nel capitolo 4 un framework di questo tipo, chiamato *OMNeT++* e ne studieremo gli aspetti principali.

3.3 La generazione dei numeri casuali

In sezione 3.1 abbiamo accennato al fatto che nei modelli stocastici le misure ottenute non possono essere usate come valori singoli ma soltanto valutate come delle variabili aleatorie, a causa di alcuni elementi di "casualità" introdotti nella simulazione. Non ci siamo però addentrati concretamente nel come sia possibile introdurre queste caratteristiche d'incertezza in una simulazione computerizzata. Tutto ciò è possibile grazie ai cosiddetti **generatori di numeri casuali**, algoritmi in grado di creare sequenze di numeri $(R_1, R_2, ...)$ che devono possedere le proprietà statistiche di *uniformità* (ciascun valore deve avere esattamente la stessa probabilità degli altri di essere osservato) e *indipendenza* (l'estrazione di un certo valore non deve essere influenzata da quelle effettuate precedentemente e non deve influenzare quelle che seguiranno). L'obiettivo è quello di costruire un generatore che segua una *distribuzione uniforme* nell'intervallo [0,1].
Quelli che in realtà vengono usati nei simulatori a eventi discreti sono dei **generatori di numeri pseudocasuali** che non creano sequenze di valori realmente random, ma sfruttano algoritmi deterministici per ottenere delle serie che imitano le caratteristiche delle controparti realmente casuali. In generale questi meccanismi necessitano di un *seme iniziale* che da l'avvio e definisce i numeri ritornati di conseguenza. Un generatore di numeri pseudocasuali dovrebbe possedere le seguenti caratteristiche:

- **Velocità**: Il metodo di selezione dei valori deve essere efficiente, siccome potrebbe dover essere applicato per molte volte durante una simulazione.
- **Portabilità**: La tecnica dovrebbe essere trasportabile tra linguaggi di programmazione e sistemi diversi in modo che il programma di simulazione possa riprodurre gli stessi risultati ovunque sia eseguito.
- **Ciclo lungo**: La *lunghezza del ciclo* rappresenta il numero di elementi della sequenza prima che il generatore cominci a ripeterli dall'inizio. Se devono essere gestiti milioni di eventi il periodo deve essere sufficientemente lungo.
- **Ripetibilità**: Questa caratteristica garantisce al generatore la riproducibilità necessaria per le simulazioni che si vanno a eseguire, in modo che conoscendo il

seme iniziale e il generatore usato sia possibile riprodurre con precisione intere esecuzioni.

- **Casualità**: Le serie di numeri generate devono possedere le caratteristiche d'uniformità e indipendenza delle sequenze realmente casuali.

Non ci addentreremo ulteriormente nelle diverse tecniche di generazione di numeri pseudocasuali disponibili siccome *OMNeT++* si occupa già di offrire un adeguato supporto implementando l'algoritmo *Mersenne Twister* (MT) [8] che ha un periodo di $2^{19937} - 1$ ed è particolarmente efficiente se confrontato agli altri generatori di numeri pseudocasuali.

3.4 La modellazione degli input

Abbiamo discusso di come la generazione di numeri pseudocasuali sia un fattore determinante nella correttezza di una simulazione, ma non abbiamo ancora chiarito come questi possano essere usati nel funzionamento del simulatore vero e proprio.
Una componente fondamentale di un simulatore a eventi discreti è quella che viene definita come **modellazione degli input**. Ad esempio nella modellazione della comunicazione su protocollo MMS potremo avere una distribuzione d'invio delle richieste di misure o una distribuzione dei tempi di servizio del server MMS. In questi casi un generatore di numeri pseudocasuali ci permette di ottenere valori che seguono una distribuzione target a partire da una distribuzione uniforme in $[0, 1]$ (usando per esempio la tecnica della *trasformazione inversa*). Scegliere una distribuzione adeguata per ogni input è un fattore determinante per la correttezza degli output restituiti dal modello e delle successive conclusioni. I passi da seguire per scegliere quale distribuzione meglio descrive i dati in esame sono i seguenti:

1. **Collezione dei dati**: Per prima cosa è necessario raccogliere dei dati reali riguardo al processo d'input che si vuole andare a modellare. Si vanno così a registrare dei campioni dei tempi di risposta o dei tassi d'invio in esame e li si analizza. Talvolta potrebbe essere impossibile effettuare queste acquisizioni di dati (per problemi di privacy o di disponibilità) e in quelle situazioni è necessario affidarsi a esperti del dominio per poter stimare un dataset realistico. Durante queste analisi preliminari è possibile incorrere in alcuni problemi:
   - *Dati inattesi*: Potremmo compiere rilevazioni che non seguono la distribuzione che ci eravamo prefigurati. Questo potrebbe richiedere la necessità di più distribuzioni diverse per la modellazione di un singolo input.
   - *Dati variabili nel tempo*: È possibile che i dati dipendano dall'istante temporale in cui sono rilevati, richiedendo così una strutturazione più complessa all'interno della simulazione così da modellare eventuali trend o fattori di stagionalità.
   - *Dati dipendenti*: Potrebbe verificarsi che le osservazioni non siano indipendenti, ma che dipendano tra di loro.
2. **Identificazione della distribuzione**: A questo punto bisogna determinare la famiglia di distribuzioni da usare e per svolgere questo compito si possono

sfruttare gli *istogrammi* in cui viene mostrata la frequenza di valori suddivisa per diversi intervalli. Nel caso di dati continui il relativo istogramma corrisponde alla *funzione di densità di probabilità* di una distribuzione teorica. Le famiglie di distribuzioni conosciute in letteratura sono state già create con l'intento di studiare processi realmente esistenti. Se ad esempio volessimo modellare la distribuzione dei chip difettosi su lotti di n componenti in un impianto che produce un pezzo fallato con probabilità p, allora dovremmo usare una **Binomiale**. Se invece avessi bisogno della distribuzione del tempo di assemblaggio di un componente che a sua volta è composta da tante componenti di processo, allora mi servirebbe una **Normale**. In ultima ipotesi potrei modellare i tempi di interarrivo dei clienti allo sportello di una banca tramite una distribuzione **Esponenziale** o se ne avessi k potrei adoperare una **Erlang** a k stadi.

3. **Parametrizzazione della distribuzione**: Una volta individuata la corretta famiglia di distribuzioni è necessario andarla a parametrizzare sulla base dei dati raccolti. Considerando un insieme di osservazioni di dimensione n $(X_1, X_2, ..., X_n)$ possiamo calcolarne la *media campionaria* $\bar{X}$ e la *varianza campionaria* S^2 definite rispettivamente nelle equazioni 3.1 e 3.2.

$$\bar{X} = \frac{\sum_{i=1}^{n} X_i}{n} \tag{3.1}$$

$$S^2 = \frac{\sum_{i=1}^{n} (X_i - \bar{X})^2}{n-1} \tag{3.2}$$

Utilizzando quindi gli stimatori calcolati possiamo ridurci da una generica famiglia di distribuzioni a una specifica (quella che effettivamente descrive l'andamento dei nostri dati). Alcuni esempi di stima di questi parametri sono visibili in Tabella 3.1. Il parametro λ dell'*esponenziale* corrisponde all'inverso della *media campionaria*, mentre valore medio e varianza della distribuzione *normale* corrispondono esattamente alle controparti campionarie.

Distribuzione	*Parametro(i)*	*Stimatore(i) consigliato(i)*
Poisson	α	$\hat{\alpha} = \bar{X}$
Esponenziale	λ	$\hat{\lambda} = \frac{1}{\bar{X}}$
Normale	μ, σ^2	$\hat{\mu} = \bar{X}$, $\hat{\sigma}^2 = S^2$ (unbiased)

Tabella 3.1: Esempi di parametri calcolati con *media campionaria* e *varianza campionaria*

4. **Valutazione del fit**: Vengono definiti come *test di bontà dell'adattamento* tutte quelle prove usate per valutare l'aderenza di un campione di dati a una determinata distribuzione teorica. Un esempio di prova di questa tipologia è il test dei **Chi-quadro** formalizzato nell'equazione 3.3.

$$\chi_0^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} \tag{3.3}$$

Le n osservazioni vengono suddivise in k intervalli e viene calcolata la sommatoria dove O_i e E_i sono rispettivamente il numero di valori osservati e il numero di valori attesi nell'intervallo *i-esimo*. Se il valore di test supera la soglia critica scelta allora viene rifiutata l'ipotesi di uguaglianza tra la distribuzione osservata e quella teorica. Nonostante test di questo tipo siano molto utili non risultano particolarmente significativi se il dataset campione non è sufficientemente popolato.

3.5 L'analisi statistica dei modelli stocastici

L'analisi dell'output è lo studio dei dati generati da una simulazione. Lo scopo di questa fase è quello di valutare le prestazioni del sistema (*performance assolute*) o comparare le prestazioni di due o più simulatori (*performance relative*). Una prima distinzione che possiamo fare sui diversi tipi di analisi del comportamento di un sistema è la seguente:

- **Analisi in transitorio**: Si conosce lo stato iniziale del sistema e la simulazione viene terminata in base a un insieme di *eventi di fine simulazione* o raggiunto un certo *tempo di simulazione*.
- **Analisi in steady-state**: Lo stato iniziale della simulazione è definito ma il sistema viene eseguito per un periodo iniziale in modo che raggiunga una condizione di stabilità tale per cui i risultati della simulazione non dipendano più dalle condizioni di partenza. Lo stato finale del sistema non è definito a priori.

Questi **indici di prestazione** si basano sulle relative **funzioni di reward** usate per calcolarli. Ne esistono due tipologie:

- **Funzioni di reward istantanee**: Si definisce un valore fisso da accumulare in occasione di determinati eventi (es. si somma 1 all'accumulatore in occasione dell'invio di un messaggio).
- **Funzioni di reward cumulative**: Definisce una funzione che dato uno stato della rete ritorna un valore che descrive una certa proprietà di quello stato e la integra nel tempo (es. numero di messaggi in coda).

Al termine della simulazione l'indice è calcolato come **reward totale accumulato** o come **reward medio accumulato** (dividendo il reward accumulato per il numero di eventi che si sono verificati o per la durata della simulazione).
Come accennato in sezione 3.3 un modello di simulazione è una trasformazione dagli input agli output (misure) e siccome alcuni input sono variabili aleatorie allora anche gli output lo saranno. Una singola esecuzione di un simulatore stocastico restituisce in output valori di variabili casuali associate alle misure oggetto di studio, quindi per poterle analizzare adeguatamente è fondamentale raccoglierne un campione statistico significativo (eseguendo più run del simulatore con sequenze di valori pseudocasuali adeguatamente generate). Sui singoli indici ottenuti possiamo quindi eseguire due tipi di analisi statistiche:

- **Stima puntuale**: Si stima il valore atteso della distribuzione ($\hat{X} \approx E(X)$).

- **Stima intervallare**: Si calcola l'*intervallo di confidenza* che indica l'accuratezza della stima puntuale fissato un dato livello di confidenza ($\hat{X} - e_1 \leq E(X) \leq \hat{X} + e_1$).

Più precisamente l'intervallo di confidenza viene definito secondo la formula 3.4 dove $\bar{X}$ è la *media campionaria*, $t_{\frac{\alpha}{2},f}$ è l'inverso della funzione di distribuzione cumulativa (CDF) per la *t-student* con α che è il livello di confidenza scelto, f il numero di gradi di libertà e infine $\hat{\sigma}(\bar{X})$ è uno stimatore della ***deviazione standard della popolazione*** definito come $\hat{\sigma}(\bar{X}) = \sqrt{\frac{S^2}{n}}$.

$$[\bar{X} - t_{\frac{\alpha}{2},f}\hat{\sigma}(\bar{X}), \bar{X} + t_{\frac{\alpha}{2},f}\hat{\sigma}(\bar{X})] \tag{3.4}$$

Un ultimo parametro utile è la *precisione relativa* calcolata come rapporto tra la semiampiezza dell'intervallo di confidenza e la media campionaria.
Oltre alle tecniche matematiche esistono anche dei metodi grafici per valutare in maniera più qualitativa le misure ottenute. Un esempio sono i **grafici scatole e baffi** (visibili in Figura 3.3) in cui il rettangolo (centrato sulla mediana e chiamato anche *scatola*) delimita dal primo al terzo quartile, mentre il segmento (o baffo) è calcolato moltiplicando la dimensione della *scatola* per 1.5. Infine attraverso dei punti vengono specificati gli *outliers* ossia quei valori al di fuori dei *baffi* che dovrebbero verificarsi più raramente.

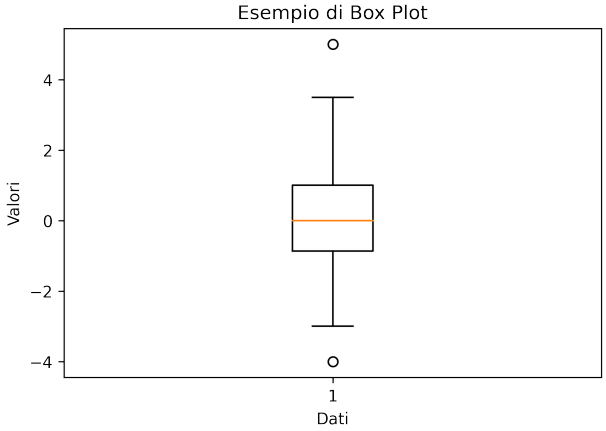


Figura 3.3: Esempio di un *Box Plot*

Altri tipi di strumenti visivi usati nelle nostre analisi sono i **grafici a dispersione** che permettono di osservare le relazioni che sussistono tra variabili quantitative (es. valore di una merce sull'asse y e tempo sull'asse x). Se consideriamo un insieme di variabili analizzate potremmo valutarne la possibile correlazione tra di esse usando un **grafico a matrice** per genera un *grafico a dispersione* per ogni coppia di variabili (sulla diagonale della matrice invece vengono mostrati degli istogrammi con la distribuzione di ciascuna variabile). Un esempio di questi ultimi grafici è visibile in Figura 3.4.

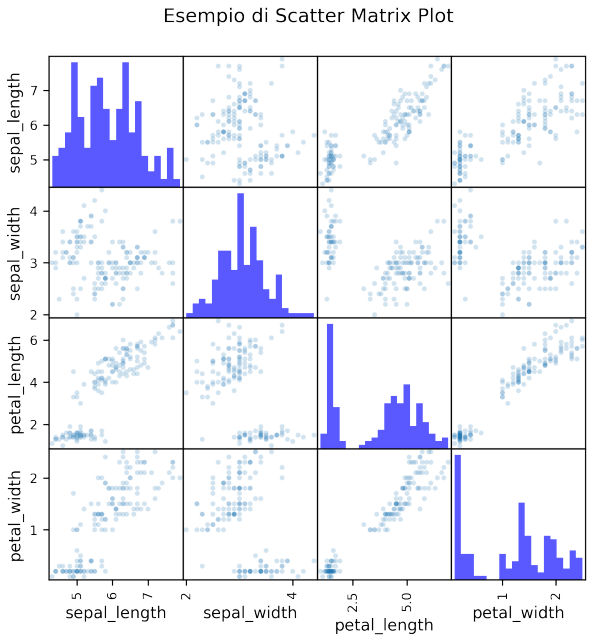


Figura 3.4: Esempio di un *Matrix Plot*

Capitolo 4

Gli strumenti utilizzati

Fino a questo punto abbiamo esplorato gli aspetti teorici della simulazione a eventi discreti e abbiamo soltanto accennato a come questi possano essere applicati in un contesto pratico tramite la loro implementazione in un linguaggio di programmazione general purpose o attraverso l'uso di librerie e framework che offrono già le basi per la realizzazione di modelli di simulazione.
In questo capitolo partiremo con l'introdurre il framework *OMNeT++*, le possibilità di modellazione che offre e i costrutti di base su cui si fonda. Analizzeremo ogni elemento nel dettaglio facendo anche un esempio pratico d'implementazione. Studieremo infine due librerie usate rispettivamente per la modellazione dello stack di rete ISO/OSI (*INET*) e delle reti 5G (*Simu5G*).

4.1 Il framework OMNeT++

OMNeT++ [10] è un framework *object-oriented* per la simulazione a eventi discreti la cui architettura modulare gli permette di essere applicato in diversi ambiti di studio: modellazione di reti cablate e wireless, modellazione di protocolli, implementazione di reti di code, modellazione di macchine multiprocessore e componentistica hardware. In generale in *OMNeT++* è possibile modellare qualunque sistema mappabile sul paradigma della simulazione a eventi discreti facendo uso di entità che comunicano tramite lo scambio di messaggi. *OMNeT++* infatti non è di per sè un simulatore in senso stretto, ma fornisce l'infrastruttura e gli strumenti per scrivere le simulazioni. I modelli possono essere assemblati a partire da componenti riutilizzabili chiamati *moduli* e possono essere connessi tra loro facendo uso di *gates* e canali, per mezzo di un linguaggio descrittivo chiamato *Network Description Language* (NED). La *Future Event List* (FEL) è implementata in *OMNeT++* usando un *heap binario* siccome tale struttura dati rende particolarmente efficiente l'operazione di estrazione del prossimo evento che deve scattare. Le simulazioni possono essere eseguite sia all'interno di un ambiente grafico chiamato *QTEnv* (utile per tenere d'occhio l'avanzamento della simulazione e fare debug in fase di sviluppo) che direttamente da linea di comando per velocizzarne il completamento. Infine questo framework fornisce quelli che sono gli strumenti per la raccolta di misure, in modo da poter tenere traccia degli eventi verificatisi durante la simulazione.

4.1.1 I componenti dei modelli

Dopo esserci fatti un'idea generale di cosa è *OMNeT++* e di quali sono le funzionalità che offre, passiamo ad analizzare i singoli componenti in maggior dettaglio.

Moduli semplici e composti

In *OMNeT++* i modelli sono strutturati attraverso una gerarchia di **moduli** complessivamente riferiti come *rete* (un esempio è visibile in Figura 4.1). Il modulo di più alto livello è definito *modulo di sistema* e può contenere diversi *sottomoduli*, i quali a loro volta ne possono contenere altri a piacimento (la profondità di questo albero non è limitata). I moduli che ne contengono altri al loro interno sono definiti **moduli composti**, mentre quelli che rappresentano le foglie della gerarchia sono chiamati **moduli semplici**. Questi ultimi sono quelli che effettivamente contengono gli algoritmi che definiscono il comportamento del modello. Sia i moduli semplici che quelli composti posseggono un *tipo* e quando questo è utilizzato come blocco costruttivo non importa che faccia riferimento a un modulo semplice o a uno composto, in questo modo è possibile dividere un modulo semplice in tanti moduli sia semplici che composti, senza influenzare gli utenti del modulo stesso.

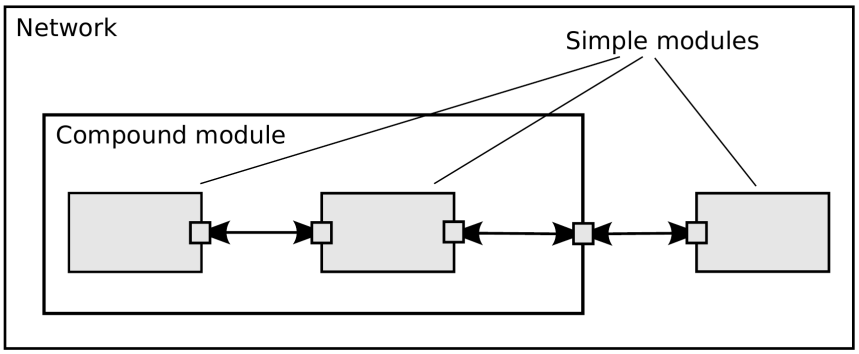


Figura 4.1: Moduli semplici e composti

Messaggi, Gates e Links

Normalmente in una simulazione a eventi discreti nella FEL vengono gestiti eventi, invece in *OMNeT++* questi elementi sono implementati tramite **messaggi**. In questo simulatore infatti i moduli comunicano tramite lo scambio di messaggi, i quali ad esempio nella simulazione di una rete informatica possono rappresentare i frame o i pacchetti, mentre nel modello di una rete di code possono simulare i clienti. I messaggi possono contenere al loro interno diversi tipi di dati (es. interi, double, array, oggetti custom), utili al destinatario per decidere le azioni da intraprendere. Gli eventi e quindi i messaggi presenti nella FEL seguono le seguenti regole:

- I messaggi vengono estratti seguendo una politica FIFO.
- Due messaggi con lo stesso tempo di arrivo vengono estratti in ordine di *priorità* decrescente.
- Due messaggi con lo stesso tempo di arrivo e la stessa priorità vengono gestiti nell'ordine in cui sono stati inseriti nella FEL.

Esistono due tipi principali di messaggi in *OMNeT++* (visibili in Figura 4.2). I primi sono i **self-messages**, messaggi che un modulo manda a se stesso quando ha la necessità di schedulare un "timer" come un evento ricorrente o un'azione che deve svolgere ma che necessita di essere ritardata. I secondi sono i **messaggi normali**, ossia quelli che vengono spediti da un modulo a un altro e servono a rappresentare la comunicazione che intercorre tra moduli diversi sia in termini di scambio di pacchetti che come scheduling di azioni (es. il componente A chiede al componente B di svolgere un compito in un certo istante di tempo).

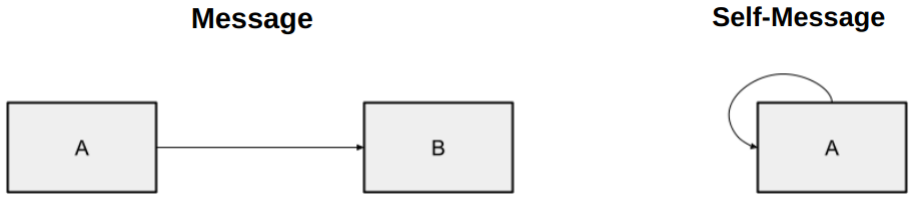


Figura 4.2: Invio di messaggi e self-messages

Ogni modulo invia e riceve i messaggi a sé destinati attraverso delle interfacce chiamate **gates** (che possono essere di *input* o di *output*). Ciascuna connessione tra due *gates* distinti è chiamata **link** e permette di collegare tra loro diversi sottomoduli o un modulo composto con gli altri al suo interno. Per rendere più agevole la modellazione della comunicazione in rete, i *link* possono essere usati per modellare i link fisici attraverso i seguenti parametri: *data rate* (velocità di trasmissione dei dati), *propagation delay* (ritardo di propagazione sul canale), *bit error rate* (errore percentuale sui singoli bit) e *packet error rate* (errore percentuale sui pacchetti). Questi parametri con i relativi algoritmi sono incapsulati negli oggetti di tipo *channel* e l'utente li può modificare in base alle sue esigenze. È anche possibile impostare il canale per recapitare un pacchetto nel momento in cui viene inviato (tutti i ritardi di comunicazione vengono annullati).

I segnali e il calcolo delle misure

Uno strumento molto potente disponibile all'interno di OMNeT++ è quello dei **segnali**. Sono un concetto molto versatile che assolve diversi compiti:

- Esporre misure utili al calcolo di statistiche del modello.
- Ricevere notifiche sui cambiamenti del modello di simulazione a *runtime*, così da poter agire di conseguenza.
- Implementare una comunicazione di tipo *publish-subscribe* tra moduli diversi (non serve che produttore e consumatore siano associati, ma si possono semplicemente modellare relazioni *many-to-many* e *many-to-one*).
- Emettere informazioni per qualunque altra ragione (es. gestione delle animazioni personalizzate visibili all'interno dell'interfaccia grafica).

La forza di questa funzionalità sta proprio nel modo in cui opera, infatti qualunque componente (modulo o canale) nella simulazione può emettere segnali e questi si propagano nella gerarchia di modulo in modulo fino alla radice. A ogni livello è

possibile registrare un listener per rimanere in ascolto su uno o più segnali, di modo che alla loro ricezione venga eseguito un metodo di *callback* che reagisca in risposta all'evento verificatosi.

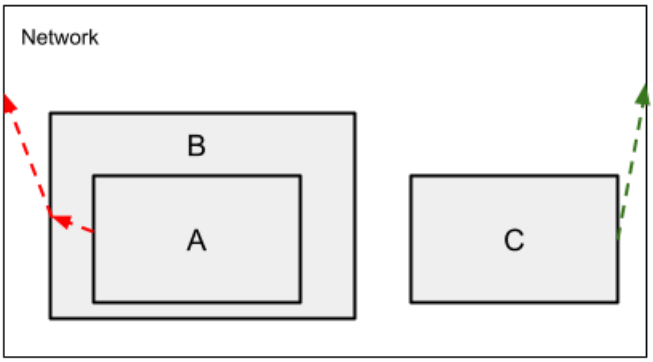


Figura 4.3: Propagazione di segnali nella gerarchia

Ad esempio osservando l'immagine 4.3, un segnale emesso dal modulo A potrà essere visto dal modulo B o dalla radice stessa, mentre non potrà essere captato dal modulo C che non è parente di A. Secondo la stessa logica un segnale emesso dal modulo C non può essere visto né da A e né da B, ma solo dalla rete al livello più alto.
Attraverso le proprietà appena espresse i segnali si rivelano un meccanismo formidabile nella registrazione di **statistiche**. Grazie a loro è infatti possibile esporre variabili senza il bisogno di specificare dove e come e debbano essere tracciate. Con questo approccio un modulo pubblica semplicemente i valori delle variabili nei diversi istanti della simulazione, ma la registrazione dei risultati vera e propria avviene nei listeners. Questi possono essere specificati sia dalla configurazione della simulazione, che dai moduli stessi. Le possibilità offerte da questa strategia ci permettono di fornire un livello di dettaglio configurabile specificando per ogni esecuzione come si vogliono registrare le variabili esposte:

- ***Scalari:*** Rappresentano valori numerici singoli ottenuti come conteggio, somma, valore minimo o valore massimo del segnale di riferimento.
- ***Vettoriali:*** Memorizzano dei vettori di valori ottenuti (solitamente su base temporale) a partire dal segnale sorgente.
- ***Istogrammi:*** Creano degli istogrammi sulla base di una proprietà definita dal segnale sorgente.

Se ad esempio vogliamo conoscere il numero di bytes inviati durante tutta la simulazione possiamo adottare *l'operatore scalare di somma* per accumulare a ogni invio la dimensione del pacchetto spedito. Se invece ci interessa conoscere tutti i valori assunti da quella variabile in un certo arco temporale, allora abbiamo bisogno di una statistica vettoriale. Questo approccio basato sui segnali ci permette anche di *modificare* e *filtrare* le misure prima che vengano registrate, ad esempio per applicare tecniche di *smoothing* o rimuovere i valori a zero. La funzionalità più utile che

possiamo sfruttare è quella di combinare le statistiche ottenendo punti di vista differenti in base al livello della gerarchia in cui queste vengono catturate. Supponendo di avere una serie di moduli identici che emettono tutti uno stesso segnale a cui è associata una certa misura (es. tempo d'invio di ciascun pacchetto), ricaveremmo statistiche diverse a seconda che agganciassimo i listener per quel segnale a livello dei singoli moduli o a livello della rete. Nel primo caso otterremmo la prospettiva dei singoli componenti (es. tempo medio d'invio dei pacchetti da parte del modulo *i-esimo*), mentre nel secondo otterremmo la prospettiva aggregata della rete (es. tempo medio complessivo d'invio dei pacchetti). Infine l'ultima funzionalità è quella di rimuovere automaticamente la parte iniziale delle misure relative al *periodo di warm-up* (durante il quale il sistema sta svolgendo operazioni d'inizializzazione).

I linguaggi NED e C++ per la topologia e la logica di simulazione

Il linguaggio **Network Description Language** (NED) viene sfruttato al fine di descrivere la struttura dei modelli di simulazione. Ciò include la definizione dei componenti presenti nella rete (moduli e sottomoduli), delle loro connessioni (tramite l'uso di *canali*) e delle relative statistiche. Le caratteristiche principali del NED sono le seguenti:

- **Struttura gerarchica**: Articolazione dei moduli composti in moduli semplici; questo permette di semplificare la realizzazione dei modelli.
- **Ereditarietà**: Le funzionalità di un modulo possono essere estese ereditando le proprietà del genitore proprio seguendo lo stile dei linguaggi di programmazione object oriented.
- **Modularità**: Tutti i moduli possono essere riutilizzati evitando di duplicare inutilmente codice e funzionalità.
- **Annotazioni con metadati**: Ogni componente può essere annotato con informazioni aggiuntive non strettamente necessarie al funzionamento della simulazione.

Il linguaggio NED possiede un'equivalente rappresentazione ad albero che può essere serializzata in XML senza perdita di dati. In questo modo viene facilitata l'estrazione automatica d'informazioni e la generazione di file NED a partire da basi di dati preesistenti (es. database SQL). Se quindi il linguaggio NED permette di definire la topologia delle reti, è il linguaggio **C++** che si occupa di strutturare la logica della simulazione. Ciascun modulo semplice ha associati i relativi file "*.cc*" e "*.h*" in cui facendo uso delle potenzialità object-oriented del linguaggio, è possibile espandere la libreria di classi di *OMNeT++* con funzionalità aggiuntive.

Gli ambienti Qtenv e Cmdenv

Una volta definiti i modelli è necessario potervi interagire in modo da riuscire a svolgere le comuni operazioni di *debug* per poterne valutarne il corretto funzionamento. Questo è lo scopo di ***Qtenv***, un'interfaccia grafica funzionante a tempo d'esecuzione che supporta la simulazione interattiva, la gestione di animazioni e il tracciamento dello stato corrente. Queste funzionalità ci permettono di seguire l'avanzamento della simulazione in ogni fase della sua esecuzione. Nelle fasi successive dell'analisi

l'obiettivo principale è quello di ottenere le misure registrate andando a eseguire più *run* di simulazione con semi d'inizializzazione dei generatori di numeri casuali diversi. Qui entra in gioco l'interfaccia da linea di comando ***Cmdenv*** che sulla base dei parametri specificati effettua l'esecuzione di batch di simulazioni. A differenza della *Qtenv* che traccia la simulazione nella sua totalità, la *Cmdenv* stampa periodicamente il numero di sequenza dell'evento corrente, il tempo di simulazione, il tempo (reale) trascorso dall'inizio della simulazione e il numero di eventi processati. La modalità di simulazione in batch permette di eseguire run diverse di simulazione in parallelo (su processi separati), ma è anche possibile sfruttare l'esecuzione distribuita nel caso i modelli fossero particolarmente complessi.

4.1.2 Un esempio pratico: Source-Sink

Nonostante abbiamo studiato tutti i componenti di *OMNeT++* dal punto di vista funzionale, non li abbiamo realmente approfonditi analizzando anche gli aspetti pratici che li caratterizzano. Per fare ciò presenteremo un esempio concreto, in modo da avere un'idea migliore di come i diversi costrutti sintattici operano nella realizzazione del modello finito. Vedremo un solo tipo di modello di base senza addentrarci ulteriormente nelle possibilità offerte dal framework e per uno sguardo più approfondito in tal senso rimandiamo al capitolo 20 di [5]. La struttura del progetto è mostrata in Figura 4.4 e si possono evidenziare alcune cartelle e file caratteristici.

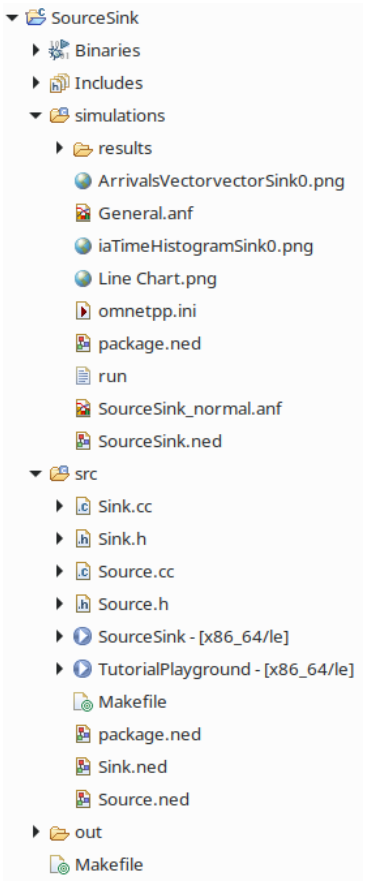


Figura 4.4: Tipica struttura di un progetto in *OMNeT++*

Nella cartella *simulations* sono contenuti solitamente i file relativi alle reti definite nella simulazione (es. *SourceSink.ned*), oltre che al file di configurazione *omnetpp.ini* e alla cartella *results* in cui il simulatore genera i file dei risultati scalari e vettoriali per ogni run. Nella cartella *src* sono presenti i file relativi ai moduli semplici e a quelli composti definiti in aggiunta a quelli offerti da *OMNeT++* (ne dettano sia la struttura che la logica). Infine nella cartella *out* sono contenuti i file binari generati in seguito alla compilazione eseguita attraverso il *Makefile*.
In particolare il modello in questione è una versione modificata del progetto **Source-Sink** in cui uno o più generatori di messaggi periodici `Source` sono collegati attraverso dei canali di comunicazione `Datarate` a uno o più componenti semplici `Sink` che eliminano i pacchetti dal sistema.

Il modulo semplice Source

Concentrandoci sul modulo semplice `Source` procederemo ora con l'analisi delle parti che lo compongono. I file sono tipici dei componenti base dei modelli in *OMNeT++*. La struttura interna del modulo è definita nel file `Source.ned` (visibile nel codice 4.1).

```
1  //
2  // Generates messages with a configurable interarrival time.
3  //
4  simple Source
5  {
6      parameters:
7          volatile double sendInterval @unit(s) = default(
               exponential(1s));
8          @display("i=block/source");
9      gates:
10         output out;
11 }
```

Codice 4.1: File `Source.ned`

La keyword **simple** indica la presenza di un modulo semplice, mentre in seguito alla direttiva **parameters** vengono elencati i parametri configurabili del modulo. In questo caso abbiamo un intervallo tra un invio e il successivo (`sendInterval`) che di default è generato usando una distribuzione esponenziale con valore medio di *1* secondo. Il fatto di associare il modificatore **volatile** a un parametro implica che il suo valore debba sempre essere valutato ogni volta che si accede a quella variabile. Ciò implica che per ogni nuovo accesso al parametro verrà estratto un nuovo valore dalla distribuzione di probabilità associata. L'unità specificata con `@unit(s)` serve solo per il calcolo delle misure a fine simulazione, mentre il comando `@display` indica un metadato usato solo nella simulazione con *Qtenv* (l'icona da usare per il modulo). Infine successivamente alla keyword **gates** vengono specificate tutte le porte associate al modulo (in questo caso ne abbiamo solo una di `output`).
Un file d'intestazione `Source.h` contiene le dichiarazioni dei metodi assieme ai campi caratteristici del modulo. Nel codice 4.2 possiamo apprezzare come ogni classe che voglia estendere le funzionalità dei moduli semplici di *OMNeT++* debba estendere la classe `cSimpleModule`, da cui eredita i metodi `initialize()` e

`handleMessage()`. Il primo metodo viene chiamato da *OMNeT++* in fase d'inizializzazione, successivamente alla creazione di tutti gli oggetti della rete. Questo garantisce l'assenza di errori di accesso a indirizzi nulli nel caso in cui un modulo facesse riferimento a un altro non ancora inizializzato (altrimenti la corretta esecuzione della simulazione dipenderebbe dall'ordine d'inizializzazione dei moduli nella rete). Il secondo metodo di quelli ereditati invece viene chiamato da *OMNeT++* quando si verifica lo scatto del messaggio passato come parametro. Oltre a un costruttore e a un decostruttore, in questa classe abbiamo bisogno di definire un puntatore della classe `cMessage` che terrà un riferimento al timer usato internamente per l'invio di messaggi.

```
1  /**
2   * Generates messages; see NED file for more info.
3   */
4  class Source : public cSimpleModule
5  {
6    private:
7      cMessage *timerMessage;
8
9    public:
10      Source();
11      virtual ~Source();
12
13   protected:
14     virtual void initialize() override;
15     virtual void handleMessage(cMessage *msg) override;
16 };
```

Codice 4.2: File `Source.h`

Se nel file d'intestazione sono presenti i prototipi dei metodi, è nel file `Source.cc` che si vanno a completare queste definizioni con il rispettivo corpo delle diverse funzioni. Nel codice 4.3 si parte con il dichiarare il modulo semplice attraverso il comando `Define_Module(Source)` e nel costruttore della classe (riga 7) si va a inizializzare l'oggetto `timerMessage` a `NULL`. Nel metodo `initialize()` (righe da 15 a 19) viene costruito un nuovo oggetto di tipo `cMessage` il quale viene schedulato come *self-message* usando il metodo `scheduleAt()`. In pratica il modulo programma un timer su se stesso che scatterà appena la simulazione avrà inizio (`simTime()` ritorna il tempo di simulazione corrente, che in fase d'inizializzazione è 0). Nel metodo `handleMessage()` (righe da 21 a 29) vengono gestiti gli scatti del timer (gli unici messaggi presenti per questo modulo) e al loro verificarsi viene creato un nuovo oggetto di tipo `cMessage` (chiamato `job`) che viene inviato sul gate "`out`" attraverso il metodo `send()`, per poi andare a schedulare nuovamente `timerMessage` per scattare. In particolare per svolgere quest'ultima operazione si somma al tempo di simulazione corrente (`simTime()`) un valore estratto dalla distribuzione esponenziale accedendo al parametro `sendInterval` del modulo. Infine nel decostruttore della classe, nel caso il timer sia ancora attivo (presente nella *FEL*), viene estratto e distrutto tramite il metodo `cancelAndDelete()`.

```
1  // Source.cc file
2
3  Define_Module(Source);
4
5  Source::Source()
6  {
7      timerMessage = NULL;
8  }
9
10 Source::~Source()
11 {
12     cancelAndDelete(timerMessage);
13 }
14
15 void Source::initialize()
16 {
17     timerMessage = new cMessage("timer");
18     scheduleAt(simTime(), timerMessage);
19 }
20
21 void Source::handleMessage(cMessage *msg)
22 {
23     ASSERT(msg==timerMessage);
24
25     cMessage *job = new cMessage("job");
26     send(job, "out");
27
28     scheduleAt(simTime()+par("sendInterval").doubleValue(),
           timerMessage);
29 }
```

Codice 4.3: File `Source.cc`

Il modulo semplice Sink

Una volta creati e inviati i messaggi devono essere eliminati dal sistema: questo è lo scopo del modulo `Sink`. La sua strutturazione è quella classica dei moduli semplici, infatti partendo dal file `Sink.ned` (visibile nel codice 4.4) notiamo la dichiarazione dell'icona da usare attraverso la direttiva `@display` (riga 13) e la definizione di un gate di `input` (riga 15). Ciò che troviamo in più rispetto al modulo `Source` è l'aggiunta di due coppie *segnale-statistica* che, come abbiamo spiegato in sezione 4.1.1, svolgono la funzione di emissione e registrazione delle misure. Iniziando dalla prima (righe 7 e 8) vediamo come sia stato definito un segnale `iaTimeSignal` che emette un valore intero relativo al ritardo tra la ricezione di un pacchetto e quello successivo, attraverso cui viene definita una statistica (`iaTimeHistogram`) che registra la distribuzione di questo segnale (in un istogramma) e il valore medio. Alle righe 10 e 11 è definito il segnale `arrivalsSignal` per cui viene emesso il valore 1 ogni volta che un messaggio viene ricevuto e questo è registrato dalla statistica

arrivalsVector che traccia la sequenza di valori emessi nel tempo all'interno di un vettore. Vedremo in seguito gli output ritornati da questi componenti.

```
1  //
2  // Consumes received messages and collects statistics
3  //
4  simple Sink
5  {
6      parameters:
7          @signal[iaTimeSignal](type=int);
8          @statistic[iaTimeHistogram](source=iaTimeSignal;
               record=mean,histogram);
9
10         @signal[arrivalsSignal](type=int);
11         @statistic[arrivalsVector](source=arrivalsSignal;
               record=last,vector; interpolationmode=none);
12
13         @display("i=block/sink");
14     gates:
15         input in;
16 }
```

Codice 4.4: File Sink.ned

Nel codice 4.5 viene presentato il file Sink.h. Senza ripeterci nel commentare i prototipi dei metodi initialize() ed handleMessage() (ereditati da cSimpleModule), possiamo concentrarci sulle variabili iaTimeSignal e arrivalsSignal che memorizzano dei riferimenti agli omonimi segnali, di modo che possano essere emessi durante la simulazione. Infine la variabile lastArrival tiene traccia del tempo di simulazione in cui è stato ricevuto l'ultimo messaggio, così da poter calcolare il tempo di inter-arrivo.

```
1  /**
2   * Message sink; see NED file for more info.
3   */
4  class Sink : public cSimpleModule {
5    private:
6      // state
7      simtime_t lastArrival;
8
9      // statistics
10     simsignal_t iaTimeSignal;
11     simsignal_t arrivalsSignal;
12
13   protected:
14     virtual void initialize() override;
15     virtual void handleMessage(cMessage *msg) override;
16 };
```

Codice 4.5: File Sink.h

Passando al file `Sink.cc` (mostrato nel codice 4.6) è facile notare come nel metodo `initialize()` in seguito all'inizializzazione a 0 della variabile `lastArrival`, vengano anche registrati i segnali `iaTimeSignal` e `arrivalsSignal` attraverso il metodo `registerSignal`. Nel metodo `handleMessage()` invece vengono gestiti gli arrivi dei messaggi sul gate di input, quindi si calcola il tempo trascorso dall'ultimo messaggio arrivato, si stampa un log sulla stream di output (`EV`) e si elimina il messaggio. Successivamente vengono emessi i segnali `iaTimeSignal` (con il delay calcolato) e `arrivalsSignal` (con il valore 1), per poi andare ad aggiornare la variabile `lastArrival` al tempo di simulazione corrente.

```
1  // Sink.cc file
2  Define_Module(Sink);
3
4  void Sink::initialize()
5  {
6      lastArrival = simTime();
7
8      iaTimeSignal = registerSignal("iaTimeSignal");
9      arrivalsSignal = registerSignal("arrivalsSignal");
10 }
11
12 void Sink::handleMessage(cMessage *msg)
13 {
14     simtime_t d = simTime() - lastArrival;
15     EV << "Received " << msg->getName() << endl;
16     delete msg;
17
18     emit(iaTimeSignal, d);
19     emit(arrivalsSignal, 1);
20
21     lastArrival = simTime();
22 }
```

Codice 4.6: File `Sink.cc`

La definizione della rete

Una volta creati i singoli componenti è necessario specificare come questi vadano a collegarsi tra loro all'interno del file di definizione della rete `SourceSink.ned`. Come accennato precedentemente in questa istanza abbiamo una serie di sorgenti che si collegano a diverse destinazioni: in particolare la `Source` *i-esima* è collegata alla `Sink` *i-esima* secondo la struttura definita nel codice 4.7.

```
1  //
2  // Sample network, consisting of 'num' sources and 'num'
       sinks.
3  //
4  network SourceSink {
5      parameters:
6          int num = default(2);
7
8          @statistic[iaTimeHistogram](source=iaTimeSignal;
               record=mean,histogram);
9      types:
10         channel Datarate extends ned.DatarateChannel {
11             datarate = 100Mbps; delay = 100ms;
12         }
13     submodules:
14         source[num]: Source;
15         sink[num]: Sink;
16     connections:
17         for i=0..(num-1) {
18             source[i].out --> Datarate --> sink[i].in;
19         }
20 }
```

Codice 4.7: Definizione della rete *SourceSink*

La keyword **network** permette di specificare il nome della rete. Al suo interno successivamente alla chiave **parameters** possono essere specificati tutti i parametri configurabili dall'utente nel file '*omnetpp.ini*', a cui viene associato un tipo (`int`, `long`, `char`, `cObject*` ecc...) e un valore di default (in questo caso è un intero che rappresenta il numero di coppie `Source-Sink` presenti). Inoltre viene definita una statistica (`iaTimeHistogram`) agganciata ai segnali `iaTimeSignal` provenienti dalle `Sink`: questo permette di registrare tutti i tempi di inter-arrivo in un istogramma unico, in modo da avere la visione globale invece che quella sui singoli moduli. Attraverso la keyword **types** possono essere definiti dei tipi di moduli parametrizzati a piacere, ad esempio da riga 10 a riga 12 viene specificato un nuovo tipo di canale (`Datarate`) che estende il tipo base `DatarateChannel` in cui viene impostato un tasso di trasmissione di $100Mbps$ e un ritardo fisso di $100ms$. Successivamente alla parola chiave **submoules** vengono indicati i moduli presenti nella rete: nel nostro caso abbiamo due vettori di moduli `Source` e `Sink` di dimensione `num`. I collegamenti tra i singoli moduli dei due array sono specificati in seguito alla parola chiave **connections**, in cui grazie a un ciclo `for` viene collegato il gate di output della `source` *i-esima* al gate di input della `sink` *i-esima*. In Figura 4.5 è mostrata la vista grafica della rete appena presentata.

Il file di configurazione *omnetpp.ini*

Abbiamo definito diversi parametri configurabili sparsi per il nostro modello ed è giunto il momento di capire come impostarli. Questo compito è affidato al file di configurazione `omnetpp.ini` visibile nel codice 4.8. Ogni configurazione è specifi-

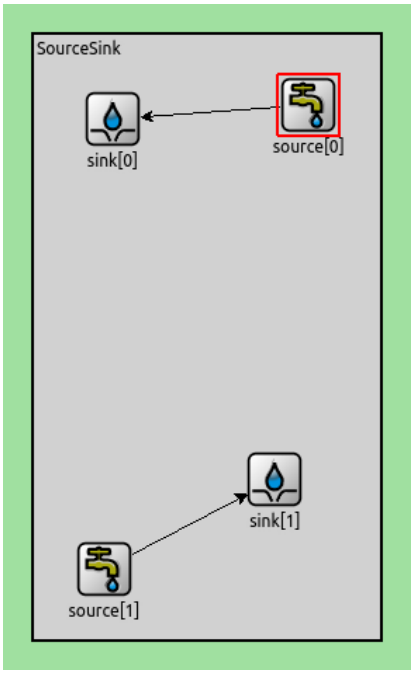


Figura 4.5: Visualizzazione in *Qtenv* della rete `SourceSink`

cata da un nome tra parentesi quadre, in particolare quella definita con il nome di `General` rappresenta una configurazione di base che viene estesa da tutte le altre configurazioni definite. Quando una configurazione *A* **estende** una configurazione *B*, *A* eredita tutti i parametri e la rete definiti in *B*. Questo riduce la quantità di codice da scrivere quando si vogliono definire configurazioni che sono variazioni minime l'una dell'altra. Quindi se in `General` viene specificata la rete `SourceSink` e viene impostato un tempo limite di simulazione di 60 secondi allora tutte le altre adotteranno queste impostazioni (a meno che non le sovrascrivano al loro interno). Se consideriamo invece la configurazione `SourceSink_uniform` (righe dalla 5 alla 7) al suo interno definisce un numero di coppie `Source-Sink` pari a 3 (con il parametro `num`) e per tutti i moduli `Source` della rete specifica una distribuzione uniforme dei tempi tra un invio e il successivo delimitata tra 1 e 10 secondi. Invece nella configurazione `SourceSink_normal` (righe da 9 a 11) vengono ereditati tutti i parametri della `SourceSink_uniform` ma in più viene indicato un tempo d'invio che segue una distribuzione normale con media di 5 secondi e deviazione standard di 2 secondi. Il fatto di poter parametrizzare tutti i moduli della rete in questo modo permette di svolgere diversi tipi di analisi senza modificare la logica di funzionamento, ma solo i parametri su cui si basa.

```
1  [General]
2  network = SourceSink
3  sim-time-limit = 60s
4
5  [SourceSink_uniform]
6  *.num = 3
7  *.source[*].sendInterval = intuniform(1s,10s)
8
9  [SourceSink_normal]
10 extends = SourceSink_uniform
11 *.source[*].sendInterval = normal(5s,2s)
```

Codice 4.8: File `omnetpp.ini` di `SourceSink`

Le statistiche ottenute

L'ultima fase dello studio è quella dell'analisi delle statistiche ottenute dalle simulazioni. Al termine della loro esecuzione nella cartella *results* vengono generati i file con le statistiche scalari (con estensione '*.sca*') e vettoriali (con estensione '*.vec*') e cliccando su uno di essi viene generato un file denominato *nome_configurazione.anf* che può essere aperto e interpretato dall'IDE per poter generare dei grafici.

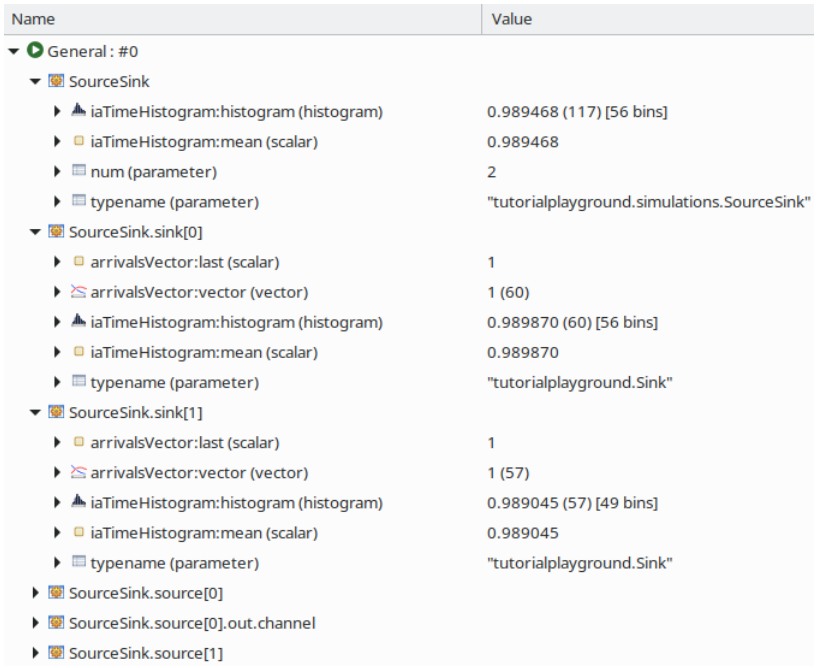


Figura 4.6: Struttura di un file *'.anf'*

In Figura 4.6 è possibile vedere la struttura di un file '*.anf*' in cui tutte le statistiche registrate sono gerarchicamente suddivise in base al modulo che le ha generate. Ad esempio possiamo notare come l'istogramma `iaTimeHistogram` e il relativo valore medio siano associati sia a ciascuna `Sink` che alla rete generale `SourceSink`. Questo perché nei relativi file NED abbiamo definito due statistiche

`iaTimeHistogram` che catturano il segnale su due livelli diversi della gerarchia e hanno quindi punti di vista differenti (uno è quello di ciascun `Sink` e l'altro è quello complessivo). Cliccando su ciascuna statistica possiamo generare il relativo grafico. Alcuni esempi di misure rilevate sul modulo `Sink[0]` in seguito all'esecuzione della configurazione `General` sono visibili nelle Figure 4.7 e 4.8. Nella prima si può notare come la distribuzione cumulativa dei tempi di inter-invio mostrata nell'istogramma sia coerente con la CDF di un'esponenziale con valore medio 1. Nella seconda invece non ci sono elementi degni di nota se non che si possono vedere graficamente gli istanti di arrivo dei messaggi al modulo `Sink[0]`.

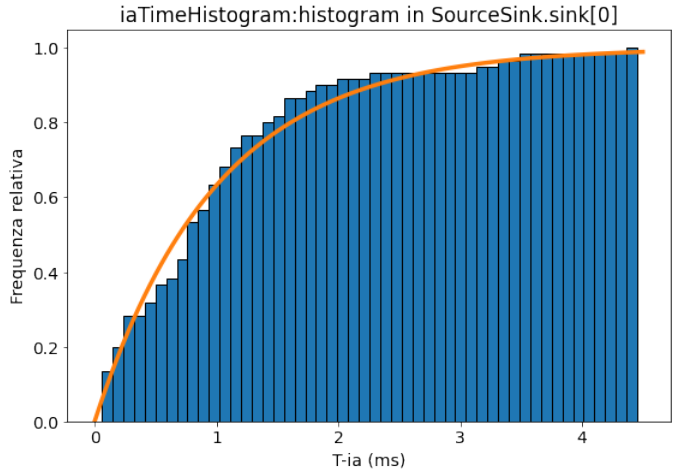


Figura 4.7: Istogramma cumulativo dei tempi di inter-invio per `Sink[0]`

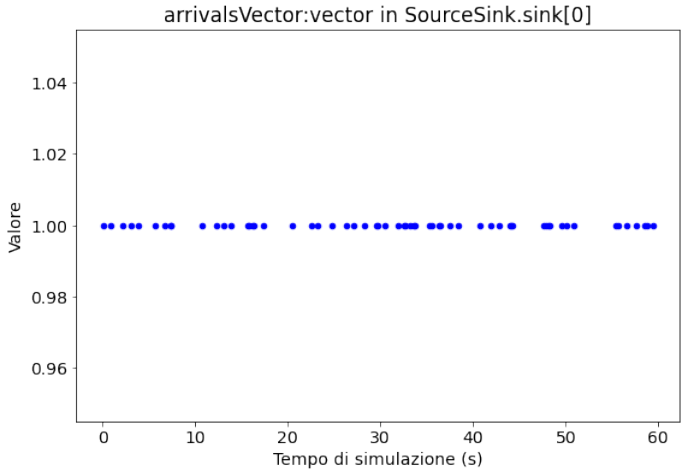


Figura 4.8: Grafico a dispersione del vettore degli arrivi per `Sink[0]`

4.2 Il framework INET

OMNeT++ è un framework particolarmente complesso che però non è stato pensato con l'idea di fornire già al suo interno tutti i moduli di cui un utente potrebbe aver bisogno, ma fa della modularità e dell'espandibilità i suoi punti di forza. Sulla base di questa premessa è stato creato ***INET*** [6], un framework open-source che sfrutta i costrutti base di *OMNeT++* (kernel di simulazione e libreria) per implementare i seguenti elementi dello stack di rete TCP/IP:

- Stack IPv4/IPv6.
- Protocolli di trasporto: TCP, UDP, SCTP.
- Protocolli di routing (ad-hoc e cablati).
- Interfacce wireless e cablate (Ethernet, PPP, IEEE 802.11, etc.).
- Livello fisico con diversi livelli di dettaglio (modelli di propagazione del segnale dallo *unit-disk* a quelli più complessi).
- Supporto a modelli di mobilità e di modellazione dell'ambiente fisico (es. ostacoli per la propagazione del segnale radio).

Agenti e protocolli di rete sono rappresentati da componenti che possono essere liberamente combinati tra loro. L'utente può programmarne di nuovi andando a studiare il funzionamento di quelli esistenti.

4.2.1 I componenti di INET

INET offre già una serie di nodi (moduli composti) preassemblati che possono essere usati per iniziare a modellare scenari di reti reali.

- **StandardHost**: Rappresenta un host della rete ed è suddiviso in una serie di livelli che seguono la gerarchia dello stack TCP/IP. Come visibile in Figura 4.9 ogni modulo di questa tipologia possiede un'interfaccia di rete di loopback (`lo[i]`) e un'interfaccia ethernet (`eth[i]`), la quale abbinata a un modulo di incapsulamento (`ethernet`) implementa il protocollo Ethernet permettendo all'host di comunicare con la rete esterna. Di default lo *StandardHost* non possiede interfacce di rete wireless ma è possibile aggiungerne con il parametro `numWlanInterfaces`, per poi farle comunicare con i livelli superiori attraverso dei moduli `ieee80211`. A livello superiore troviamo un gestore di datagrammi a livello di rete (`ipv4` o `ipv6`) e un modulo che implementa un protocollo di trasporto (`TCP`, `UDP` o `SCTP`). Infine a livello applicativo troviamo un vettore di `IApp` di dimensione `numApps` in cui possiamo specificare tutti i protocolli di alto livello che vogliamo includere (es. HTTP), ma vedremo meglio in seguito come l'azione su queste varie parti sia fondamentale per i nostri scopi.
- **WirelessHost**: È un componente analogo allo StandardHost ma presenta un'interfaccia di rete aggiuntiva Wi-Fi IEEE 802.11.
- **Switch**: Modella il comportamento di uno switch Ethernet.

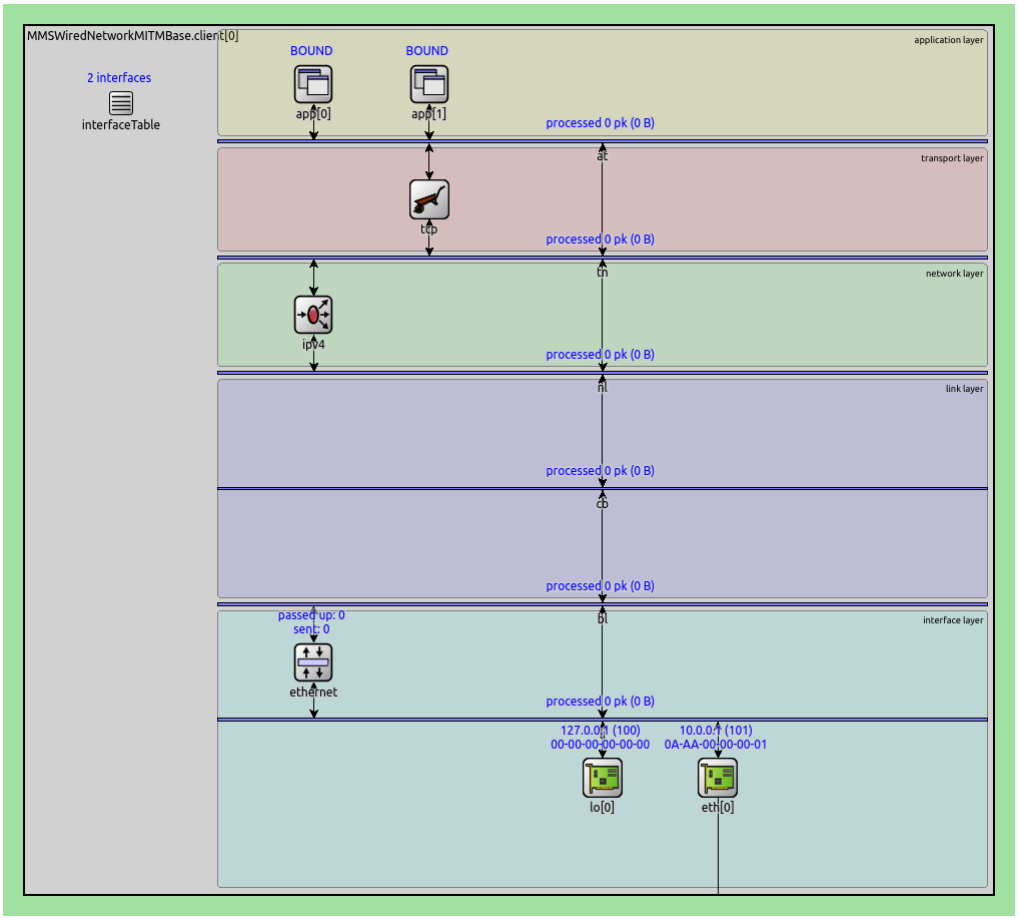


Figura 4.9: Struttura dello StandardHost

- **Router**: Svolge i compiti di un router e di default possiede una o più interfacce Ethernet (`eth[i]`) e una o più interfacce Point-to-Point (`PPP`), ma non contiene schede Wi-Fi. All'interno della `interfaceTable` sono memorizzate le associazioni tra le diverse interfacce registrate dal corrispondente modulo L2.

- **AccessPoint**: Fa le veci di un generico access point supportando più interfacce radio contemporaneamente e più porte ethernet all'occorrenza. Le operazioni di channel-scanning, autenticazione e associazione con i WirelessHost sono eseguite dalle componenti di gestione delle interfacce wlan (`Ieee80211MgmtAp`).

- **InternetCloud**: Per la maggior parte questo modulo composto è identico a un Router, ma presenta alcuni componenti aggiuntivi chiamati *delayer* (`ipv4Delayer` o `ipv6Delayer`), i quali possono essere configurati per ritardare o eliminare pacchetti in base all'interfaccia di rete da cui arrivano e a quella su cui sono diretti. Ciascun delayer deve implementare l'interfaccia `ICloudDelayer` e un esempio di ciò è il `MatrixCloudDelayer`, in cui possiamo impostare delle regole di ritardo o eliminazione, tramite l'uso di un apposito file *XML*.

```
1     <internetCloud symmetric="false">
2      <parameters>
3        <traffic src="src_host*" dest="dest_host*" delay=
            "0.012s+exponential(0.2s)" datarate="uniform
            (100kbps,1Mbps)" drop="uniform(0,1) < 0.005" /
            >
4        <traffic src="hosta" dest="hostb" delay="0.012s+
            exponential(0.2s)" datarate="uniform(500kbps,1
            Mbps)" drop="uniform(0,1) < 0.005"/>
5        <traffic src="hostb" dest="hosta" delay="0.010s+
            exponential(0.2s)" datarate="uniform(100kbps
            ,500kbps)" drop="uniform(0,1) < 0.003"/>
6        <traffic src="*" dest="*" delay="0.02s+
            exponential(0.5s)" datarate="uniform(100kbps
            ,200kbps)" drop="uniform(0,1) < 0.03" />
7      </parameters>
8    </internetCloud>
```

Codice 4.9: Esempio di configurazione di un cloud delayer

Nell'esempio visibile nel codice 4.9 per prima cosa alla riga 1 viene specificato se le regole debbano essere di default simmetriche o meno (valide in entrambe le direzioni). Dopodiché per ogni regola definita con il tag `traffic` vene indicato un host sorgente (`src`), un host destinazione (`dest`), un datarate del canale su cui viaggerà il pacchetto, il ritardo fisso da aggiungere e un valore booleano che simuli il drop di un pacchetto. Gli attributi sorgente e destinazione possono contenere wildcard per fare riferimento a più moduli contemporaneamente, mentre gli attributi `datarate`, `delay` e `drop` possono essere specificati sotto forma di espressioni NED che verranno valutate per ogni pacchetto ricevuto. Questo modulo `InterneCloud` ci permette di simulare l'azione che una rete esterna avrebbe sui pacchetti che la attraversano e ciò è utile per simulare il comportamento *best-effort* di una simile architettura.

- **Network Configurator**: I moduli di questo tipo hanno il compito di svolgere la configurazione iniziale della rete, che può essere eseguita sia in maniera manuale che completamente automatica. In prima battuta, dopo aver costruito un grafo pesato contenente la tipologia di rete, avviene l'assegnazione degli IP sulle varie interfacce dei moduli. Nello specifico durante la generazione del grafo viene assegnato peso infinito ai nodi per cui il `forwarding` non è abilitato (es. gli host), così da non farvi transitare pacchetti non diretti a loro, mentre ai router è assegnato peso 0. Agli archi è invece assegnato un peso inversamente proporzionale al loro datatarete. Tutto ciò per permettere all'algoritmo di routing *Weighted Shortest Path* di riempire e in seguito ottimizzare le entry nelle tabelle di routing. Infatti subito dopo l'esecuzione dell'algoritmo le tabelle d'instradamento conterranno una entry per ogni interfaccia nella rete, ma molte di queste possono essere fuse in base alla sottorete di appartenenza, andando a minimizzarne il numero.

- **ScenarioManager**: In alcune situazioni può rivelarsi utile simulare lo spegnimento di un modulo o la disconnessione di un canale così da simulare un eventuale crash di un router o interruzione della connessione. A questo scopo è possibile introdurre nei modelli un modulo di tipo `ScenarioManager` che in base al file *XML* che riceve in input ci permette di gestire diversi aspetti della rete. Un esempio di questi script è visibile nel codice 4.10 dove all'interno del tag `scenario` definiamo che a 50 secondi dall'inizio della simulazione il `router[2]` verrà spento (con il tag `shutdown`) e disconnesso dalla rete pruncipale (vengono scollegati i gates `eth[0]` ed `eth[1]` nel tag `disconnect`).

```
1 <scenario>
2   <at t="50">
3     <disconnect src-module="router[2]" src-gate="ethg[0]
        "/>
4     <disconnect src-module="router[2]" src-gate="ethg[1]
        "/>
5     <shutdown module="router[2]"/>
6   </at>
7 </scenario>
```

Codice 4.10: Esempio di file *XML* per configurare uno scenario

Un elemento fondamentale specialmente nella modellazione delle reti wireless è il **modulo di mobilità**. Ogni host nella rete ne possiede uno e il suo scopo è quello di definire posizione, velocità e accelerazione del proprietario su un piano Euclideo tridimensionale in ogni istante della simulazione. Questo permette di modellare fenomeni come la potenza del segnale ricevuto, interferenze, saturazione del canale e in generale tutte quelle caratteristiche che sono influenzate dalla distanza che i nodi hanno tra loro. Questi moduli sono solitamente implementati tramite algoritmi in C++ e quasi tutti posseggono dei parametri che definiscono la posizione iniziale del componente a cui sono associati. Possiamo suddividere i modelli di mobilità in cinque categorie principali:

1. **Stazionari**: Permettono di definire soltanto la posizione e l'orientamento di chi li possiede ma non il movimento. Quello principalmente usato nelle nostre simulazioni è lo `StationaryMobility` che fornisce solo posizionamento deterministico e random.

2. **Deterministici**: Sfruttano modelli matematici non casuali per descrivere il movimento. Sono stati fatti alcuni test con il `LinearMobiliy` che simula un movimento lineare a velocità o accelerazione costante, ma ne esistono anche altri che simulano il movimento circolare e su un rettangolo come il `CircleMobility` e il `RectangleMobility`.

3. **Stocastici**: Si basano su modelli matematici che sfruttano i numeri random per il loro funzionamento. Possono basarsi su una sequenza di numeri pseudocasuali vera e propria oppure su una matrice di transizione probabilistica per il cambio di stato (es. fermo o in movimento).

4. **Basati su traccia**: Seguono il movimento registrato da parte di un oggetto reale e a seconda del tipo di modulo usato la traccia da emulare può avere diversi formati.

5. **Combinati**: Non sono di per sé dei veri e propri modelli di mobilità, ma permettono di combinarne diversi al loro interno, in modo da ottenerne movimenti più complessi.

Un ulteriore componente richiesto nella simulazione delle comunicazioni wireless è quello che si occupa di modellare il **mezzo di trasmissione** condiviso dove avviene la comunicazione. Questo tiene traccia di trasmettitori, ricevitori, sorgenti di rumore, trasmissioni in atto e rumore di fondo. Ogni modello di mezzo di trasmissione che si definisca tale deve possedere al suo interno i seguenti sottomodelli: ***propagazione del segnale*** (descrive come il segnale si muove nello spazio e calcola le coordinae spazio-temporali dei moduli che ricevono), ***path loss*** (modellano come la potenza del segnale diminuisce in base a fattori come distanza, rifrazione, riflessione e assorbimento), ***obstacle loss*** (spiega come il segnale si degradi propagandosi attraverso oggetti fisici fatti di materiali diversi), ***background noise*** (include la modellazione di fenomeni come rumore termico, rumore cosmico di fondo e fluttuazioni elettromagnetiche casuali che influenzano il segnale, ma che non provengono da una sorgente specifica, quindi non avrebbe senso modellarne la propagazione nello spazio e nel tempo) e ***segnale analogico*** (sceglie come il segnale analogico debba essere rappresentato). In *INET* è possibile scegliere il livello di dettaglio da adottare per ciascuno dei sottomoduli sopra elencati, oppure si possono adottare dei modelli di comunicazione preconfigurati come quelli che seguono:

- **`UnitDiskRadio`**: Fornisce un livello fisico semplificato ma molto efficiente e il cui comportamento è piuttosto facile da predire. L'idea dietro questo modello di trasmissione radio è quella di descrivere la comunicazione con pochi parametri basati sulla distanza come range di comunicazione, range d'interferenza, e range di rilevamento. Che la ricezione di un messaggio abbia successo o meno dipende esclusivamente dalla distanza del ricevitore dal trasmettitore. In questo modello è anche supportata la comunicazione su canale ideale dove tutte le trasmissioni sono ricevute con successo indipendentemente dalla distanza o dalle interferenze.

- **`Ieee80211ScalarRadio`**: Si basa sul modello scalare di calcolo della potenza di trasmissione (in altre parole la potenza del segnale non cambia in base al tempo o alla frequenza) seguendo la rappresentazione analogica del segnale. Questo significa che valori come il range di trasmissione non saranno direttamente impostabili, ma dipenderanno a loro volta da altri parametri come la potenza di trasmissione (espressa in $mWatt$).

4.2.2 Le TCPApplications

Il fulcro del protocollo MMS che ci interessa implementare risiede nel livello applicativo dello stack di rete. Questo significa dover sviluppare un modulo che estenda l'interfaccia `IApp` di *INET* e il cui tipo possa così essere usato nel vettore `app` di uno `StandardHost` o di un `WirelessHost`. Scrivere da zero un componente simile è sicuramente un compito fattibile, ma ciò che conviene fare è estendere i seguenti

moduli messi a disposizione dalla libreria per lo sviluppo di un generico protocollo di comunicazione *request-response* basato su TCP:

- **`TcpBasicClientApp`**: Base per il componente client che comunica con il server per mezzo di sessioni. Durante una sessione il client apre una singola connessione TCP con il server, invia diverse richieste (attende sempre che venga ricevuta la risposta completa) e chiude la connessione. Un vincolo da tenere presente di questa implementazione è che ogni client può connettersi al massimo con uno e un solo server in un certo istante della simulazione. Come visibile nel codice 4.11 nella sezione `parameters`, l'indirizzo di connessione viene impostato nel parametro `connectAddress`, mentre i parametri `startTime` ed `endTime` memorizzano il tempo d'inizio della sessione e quello di fine invio. Il numero di richieste da inviare in ogni sessione e la lunghezza di richiesta e risposta sono configurabili nei parametri `numRequestsPerSession`, `requestLength`, `replyLength`. Il parametro `thinkTime` contiene il gap temporale tra richieste, mentre `idleInterval` e `connectInterval` contengono rispettivamente il tempo che deve trascorrere tra richieste consecutive e tra sessioni diverse. Infine vengono dichiarati due segnali (`packetSent` e `packetReceived`) in cui vengono emessi i pacchetti inviati (il tipo è `inet::Packet`). Sulla base di questi segnali vengono generate diverse statistiche come `packetSent` e `packetReceived` che tracciano la dimensione e il numero di pacchetti spediti sia sotto forma di scalari che di vettori. Infine altre statistiche tengono conto del numero di sessioni attive e del numero di sessioni totali. La prima delle due in particolare memorizza il valore massimo, la media temporalmente pesata e il vettore di valori, mentre il secondo calcola solo la somma totale. Nella parte finale del file sono dichiarati i `gates` di input e di output che permettono all'applicazione di comunicare con i livelli sottostanti della rete.

```
1  simple TcpBasicClientApp like IApp
2  {
3      parameters:
4          string localAddress = default("");
5          int localPort = default(-1);
6          string connectAddress = default("");
7          int connectPort = default(1000);
8          double startTime @unit(s) = default(1s);
9          double stopTime @unit(s) = default(-1s);
10         volatile int numRequestsPerSession = default(1);
11         volatile int requestLength @unit(B) = default
               (200B);
12         volatile int replyLength @unit(B) = default(1MiB
               );
13         volatile double thinkTime @unit(s);
14         volatile double idleInterval @unit(s);
15         volatile double reconnectInterval @unit(s) =
               default(30s);
16         int timeToLive = default(-1);
17         int dscp = default(-1);
```

```
18          int tos = default(-1);
19          @display("i=block/app");
20          @lifecycleSupport;
21          double stopOperationExtraTime @unit(s) = default
                (-1s);
22          double stopOperationTimeout @unit(s) = default(2
                s);
23          @signal[packetSent](type=inet::Packet);
24          @signal[packetReceived](type=inet::Packet);
25          @signal[connect](type=long);
26          @statistic[packetReceived](title="packets
                received"; source=packetReceived; record=
                count,"sum(packetBytes)","vector(packetBytes)
                "; interpolationmode=none);
27          @statistic[packetSent](title="packets sent";
                source=packetSent; record=count,"sum(
                packetBytes)","vector(packetBytes)";
                interpolationmode=none);
28          @statistic[endToEndDelay](title="end-to-end
                delay"; source="dataAge(packetReceived)";
                unit=s; record=histogram,weightedHistogram,
                vector; interpolationmode=none);
29          @statistic[numActiveSessions](title="number of
                active sessions"; source=warmup(sum(connect))
                ; record=max,timeavg,vector;
                interpolationmode=sample-hold;
                autoWarmupFilter=false);
30          @statistic[numSessions](title="total number of
                sessions"; source="sum(connect+1)/2"; record=
                last);
31      gates:
32          input socketIn @labels(TcpCommand/up);
33          output socketOut @labels(TcpCommand/down);
34  }
```

Codice 4.11: File `TcpBasicClientApp.ned`

- **`TcpGenericServerApp`**: Base per il componente server che riceve le connessioni da parte dei client. Questo modulo accetta un qualunque numero di connessioni TCP e si aspetta di ricevere messaggi della classe `GenericAppMsg`. In risposta il server manda indietro al mittente lo stesso messaggio cambiandone solo la dimensione in base al valore specificato dal client nel pacchetto. Come si può vedere nel codice 4.12 il ritardo con cui il server invia la risposta può essere definito nel parametro `replyDelay`. Oltre ai segnali e alle statistiche analoghe a quelle presenti nel client ne è stata introdotta una aggiuntiva che ritorna la distribuzione e i singoli valori di ritardo complessivo che il pacchetto ha impiegato per raggiungere il server.

```
1  simple TcpGenericServerApp like IApp
2  {
3      parameters:
4          string localAddress = default("");
5          int localPort = default(1000);
6          double replyDelay @unit(s) = default(0s);
7          @display("i=block/app");
8          @lifecycleSupport;
9          double stopOperationExtraTime @unit(s) = default
               (-1s);
10         double stopOperationTimeout @unit(s) = default(2
               s);
11         @signal[packetSent](type=inet::Packet);
12         @signal[packetReceived](type=inet::Packet);
13         @statistic[packetReceived](title="packets
               received"; source=packetReceived; record=
               count,"sum(packetBytes)","vector(packetBytes)
               "; interpolationmode=none);
14         @statistic[packetSent](title="packets sent";
               source=packetSent; record=count,"sum(
               packetBytes)","vector(packetBytes)";
               interpolationmode=none);
15         @statistic[endToEndDelay](title="end-to-end
               delay"; source="dataAge(packetReceived)";
               unit=s; record=histogram,weightedHistogram,
               vector; interpolationmode=none);
16     gates:
17         input socketIn @labels(TcpCommand/up);
18         output socketOut @labels(TcpCommand/down);
19 }
```

Codice 4.12: File `TcpGenericServerApp.ned`

- **`GenericAppMsg`**: È il tipo del messaggio che client e server si scambiano e la sua struttura è visibile nel codice 4.13. Il messaggio è costruito al di sopra di un'altra struttura dati chiamata `FieldChunk`, usata per rappresentare pezzi di dati suddivisi in campi predefiniti. Nel parametro `expectedReplyLength` è specificata la dimensione attesa del pacchetto di risposta, mentre in `replyDelay` viene indicato il ritardo con cui inviare la risposta. Infine nel campo `serverClose` un valore booleano impartisce al server un eventuale comando di chiusura della connessione in seguito all'invio della risposta. In fase di compilazione *OMNeT++* converte i file con estensione "`.msg`" nelle rispettive definizioni in C++, generando automaticamente costruttori, descostruttori, getter, setter e tutte le funzioni necessaire al framework per la loro integrazione.

```
1 class GenericAppMsg extends FieldsChunk
2 {
3     B expectedReplyLength;
4     double replyDelay;
5     bool serverClose;
6 }
```

Codice 4.13: File `GenericAppMsg.msg`

Non ci addentreremo adesso nei dettagli dell'implementazione che caratterizzano la parte in C++, siccome verranno meglio spiegati in sezione 5.2 nella loro versione definitiva usata per la realizzazione di MMS.

4.3 La libreria Simu5G

Un obiettivo delle nostre prove era quello di testare il funzionamento dei nostri attacchi e protocolli su reti che sfruttassero tecnologie di comunicazione altamente affidabili e a banda elevata, senza la necessità di un'infrastruttura completamente cablata. In questo contesto abbiamo voluto adottare il 5G come sistema di comunicazione con i server DER, per poter modellare un ambiente realistico in cui gli endpoint spesso si appoggiano a sistemi New Radio (NR) 3GPP per poter funzionare. Seguendo questo approccio abbiamo scelto ***Simu5G*** [9], un framework di simulazione delle *5G Radio Access Network* (RAN) basato sulla libreria di simulazione di reti 4G *SimuLTE* (di cui mantiene buona parte della struttura). Questo framework si fonda anch'esso sui componenti offerti da *OMNeT++* e fornisce una collezione di modelli con interfacce ben definite che possono essere istanziati e connessi per modellare scenari arbitrariamente complessi. In *Simu5G* sono stati incorporati molti dei modelli di *INET* per permettere di integrare le reti 5G con i restanti componenti delle reti TCP/IP simulate. Il focus principale di questa libreria è quello di simulare il *data-plane* del 5G (trasmissione dati) lasciando espressa ad alto livello la parte di *control-plane* (operazioni di controllo e configurazione).

4.3.1 I componenti di Simu5G

Per poter realizzare le simulazioni di queste reti in *Simu5G* sono stati introdotti nuovi moduli composti con funzionalità ben specifiche:

- **NR User Equipment (NrUE)**: Rappresenta qualunque dispositivo in grado di connettersi a una rete 5G. Possiamo pensarlo come l'equivalente di un Wireless Host o di uno Standard Host di *INET* ma fornito di un'interfaccia `NRNicUe` con doppio stack per connettersi a queste nuove reti mobili (sia LTE che 5G) invece che agli Access-Point Wi-Fi. In maniera simile possiamo quindi specificare il tipo delle nostre applicazioni nel vettore `app` del modulo stesso. Supporta l'associazione dinamica alla migliore gNB nelle vicinanze, ma è anche possibile configurare questo aspetto manualmente da file "*.ini*".
- **G-Node-B (gNB)**: Anche definite come *Base Stations* rappresentano il limite tra la *Radio Area Network* (RAN) e la *Core Network*, grazie alle quali

i dispositivi UE possono collegarsi all'infrastruttura cablata attraverso la loro interfaccia `NrNicGnb`. Queste gNB comunicano le une con le altre attraverso delle interfacce X2, con cui implementano le operazioni di handover per il passaggio di un UE da una base station a un'altra. Come è possibile vedere in Figura 4.10 si può sia realizzare uno *Standalone* (SA) deployment, dove è presente solo connettività 5G, che una *Dual Connectivity*, dove connessione LTE e 5G coesistono e la gNB funziona come Nodo Secondario (SN) in favore della eNB LTE che è il Nodo Master (MN).

- **User Plane Function (UPF)**: Rappresenta il limite tra la *Core Network* e il resto della rete. Il compito principale di questo componente è quello di attuare il forwarding dei pacchetti provenienti dall'esterno verso la *Core Network* attraverso il *GPRS Tunneling Protocol* (GTP) e grazie a un `filterGate` con cui si connette verso la rete principale. Un'altra funzionalità delle UPF è quella di poter agire come *fog nodes* su cui eseguire calcoli distribuiti e comunicare tempestivamente i risultati ai dispositivi connessi sul territorio.

- **Binder**: È un modulo visibile da tutti gli altri nodi nella rete ed ha il compito di memorizzare informazioni sulle componenti 5G (es. un riferimento). Svolge il ruolo di "oracolo" che implementa tutte le funzioni di control-plane, senza la necessità di un protocollo di gestione ad-hoc. Un esempio di utilizzo di questo modulo riguarda il calcolo del *Signal-to-Noise-plus-Interference Ratio* (SINR) che determina il rapporto tra la potenza del segnale d'interesse e la potenza del rumore di fondo sommato all'interferenza di altre trasmissioni sullo stesso canale (informazione ottenibile solo grazie al modulo binder).

- **Carrier Aggregation**: Modulo globale che memorizza tutte le informazioni relative ai *Carrier Components* (CCs) impiegati nella rete. Questi elementi rappresentano porzioni disgiunte della banda di frequenza e sono usate in gruppi simultaneamente nelle comunicazioni 5G. Questo modulo di *Carrier Aggregation* contiene un vettore di N sottomoduli *componentCarrier* ognuno dei quali è configurabile sotto l'aspetto della banda di frequenza a cui è associato. Ogni coppia UE/gNB potrebbe essere in grado di usare solo un sottoinsieme dei CCs disponibili e la comunicazione è possibile solo se entrambi i componenti supportano almeno un CC in comune.

Vedremo meglio nel capitolo 5 come sono stati integrati questi moduli nei diversi modelli realizzati.

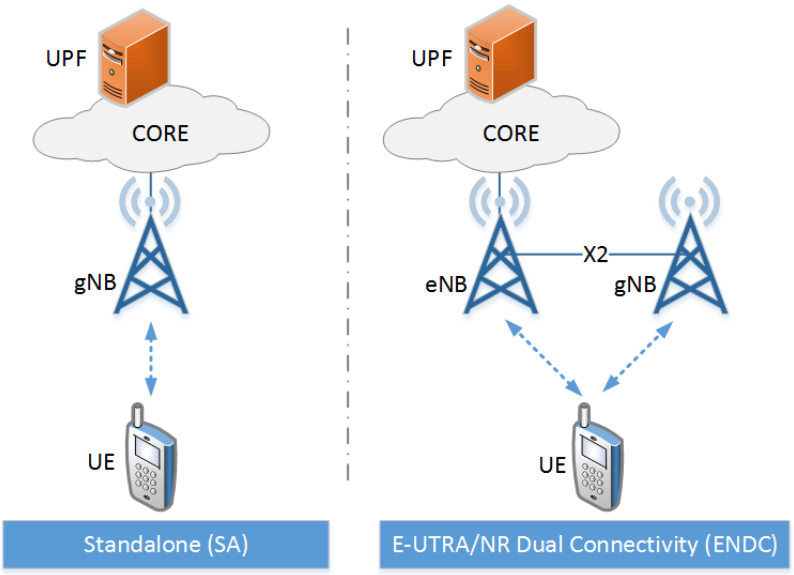


Figura 4.10: Architettura di una rete 5G in versione Standalone (sinistra) e Dual Connectivity (destra)

Capitolo 5

I modelli

Una volta appresi quelli che sono gli strumenti necessari per la realizzazione delle nostre simulazioni, possiamo finalmente addentrarci in quelli che sono i modelli oggetto del nostro lavoro. Iniziando dal comprendere le scelte fatte nella progettazione del modello MMS a partire dalla specifica del protocollo, passeremo all'analisi di tutti i moduli che lo compongono. Dai campi che formano il messaggio MMS, ci sposteremo a commentare la struttura delle applicazioni che compongono client e server, per poi concludere con le varianti introdotte per la realizzazione degli attacchi di *Distributed Denial of Service* e *Man In The Middle*. Infine ci concentreremo sull'analisi dei modelli veri e propri con le rispettive topologie di rete e configurazioni.

5.1 La fase di progettazione

Nella creazione di un modello che imiti il comportamento del relativo protocollo simulato è necessario operare delle scelte che ne definiscano la complessità e il livello di dettaglio. Lo scopo principale del nostro studio è quello di poter osservare le interazioni che avvengono tra un client e un server MMS sia all'interno di scenari diversi (es. topologie di rete e attacchi di vario tipo) che permettendo d'impostare parametri diversi in base alla situazione da simulare (es. variare il tempo d'invio del client o il tempo di risposta del server). Ciò implica che dove sarà necessario decidere preferiremo sempre avere una maggiore capacità di parametrizzazione dei modelli rispetto ad una perfetta corrispondenza con il protocollo reale. Non ci interessa emulare l'esatto comportamento che otterremmo nella corrispettiva situazione reale, pertanto vengono introdotte alcune astrazioni: ci importa ottenere uno strumento il più possibile configurabile e versatile, evitando di appesantirlo con dettagli che farebbero aumentare il costo delle simulazioni. In questo modo abbiamo la possibilità di provare a simulare situazioni limite che altrimenti non ci sarebbe possibile sperimentare.

5.2 Il modello MMS

Abbiamo già citato in sezione 2.2 quali funzionalità del protocollo abbiamo scelto d'introdurre, ma non abbiamo ancora visto tutto ciò nell'ottica di doverlo implementare nel framework `OMNeT++`. Guardando al funzionamento regolare della comunicazione, il client in seguito all'avvio dovrà collegarsi con ciascun server presente nella

topologia (o a quelli previsti in base alla sua configurazione) e a ciascuno invierà un messaggio di sottoscrizione per la ricezione di *InformationReport* (misure inviate periodicamente con regolarità dai server MMS). Dopodiché in base a diversi intervalli di tempo configurabili il client invia richieste di *lettura di misure* o *esecuzione di comandi* a cui il server risponde con il valore richiesto o con un acknowledgement sull'esito dell'operazione eseguita. Passando ora alla fase di operatività anomala, per la realizzazione di un attacco di *Distributed Denial of Service* è sufficiente implementare una variante del client in cui dopo essersi connesso ai server MMS compromessi, inizierà a inviare a loro richieste di lettura o esecuzione di comandi false e a un tasso che rende impossibile al server gestire gli altri messaggi MMS in arrivo. Guardando invece all'attacco *Man In The Middle* abbiamo la necessità di un componente più complesso che si frapponga nella connessione tra client e server MMS e, potendo leggere il contenuto di ogni messaggio MMS inviato in entrambe le direzioni, possa decidere se lasciarlo passare, modificarlo o bloccarlo completamente. Procediamo ora passo dopo passo a presentare tutti i componenti necessari per realizzare questi scenari.

5.2.1 La definizione del messaggio

Prima di definire i singoli componenti dell'architettura incominciamo dall'esporre la composizione dei messaggi che essi si scambiano. Nel file `MmsMessage.msg` visibile nel codice 5.1 è possibile notare come nella parte iniziale siano definiti diversi *enumeratori* (`enum`), ognuno dei quali identifica una specifica caratteristica del pacchetto MMS. L'enumeratore `MMSKind` indica il tipo generale del messaggio: se rappresenta una comunicazione malevola nel caso di un attacco DDoS (`FAKE`), se è un messaggio di sottoscrizione di un client alla ricezione di misure periodiche (`CONNECT`) oppure se rappresenta una misura, una richiesta generica o una risposta generica (`MEASURE`, `GENREQ` o `GENRES`). Nell'enumeratore `ReqResKind` troviamo l'indicazione del fatto che il messaggio sia di lettura di una misura (`READ`) o d'invio di un comando (`COMMAND`) nel caso il tipo generico faccia riferimento a una richiesta (`GENREQ`) o a una risposta (`GENRESP`). Se invece un pacchetto è di tipo misura (`MEASURE`) il suo `ReqResKind` sarà impostato ad `UNSET`. In realtà questa distinzione tra richieste e risposte sarebbe superflua siccome tale informazione può essere determinata sulla base della direzione del pacchetto (se si dirige da client a server o viceversa). Questa però è stata comunque inclusa in modo da facilitare la lettura e la gestione dei log, pur non influenzando i tempi di trasmissione. L'ultimo enumeratore definito viene usato negli attacchi Man In The Middle per determinare se un messaggio è stato compromesso (`COMPR`), bloccato (`BLOCK`), generato dall'attaccante (`FAKEGEN`) o se non è stato attaccato in alcun modo (`UNMOD`). Anche in questo caso stiamo introducendo un campo aggiuntivo che rappresenta più un metadato del pacchetto e normalmente non sarebbe disponibile nel protocollo reale. Questo è però fondamentale dal momento che stiamo modellando uno scenario con un livello di astrazione tale da non includere gli elementi che ci servirebbero per poter definire condizioni di normalità e anormalità (es. variazione di una misura nel tempo o esecuzione realistica di comandi). La parte principale del file è quella in cui avviene la definizione della classe `MmsMessage` come estensione di `FieldsChunk`. Al suo interno troviamo un campo `originId` che permette di fare l'associazione tra richieste e risposte, due campi `expectedReplyLength` e `serverClose` con la stessa funzione che

avevano nel `GenericAppMsg` presentato in sezione 4.2.2 e una variabile per ciascuno degli enumeratori elencati in precedenza (`reqResKind`, `messageKind` e `atkStatus`). Infine sono stati inseriti alcuni campi necessari per la modellazione dell'attacco Man In The Middle (`evilServerConnId` e `serverIndex`), il cui significato sarà meglio chiarito in seguito, e un campo `data` per eventuali espansioni future sulla rappresentazione di dati nei messaggi MMS.

```
1  enum MMSKind {
2      // Value used by the BadMmsClient
3      FAKE = -1;
4
5      CONNECT = 0;
6      MEASURE = 1;
7      GENREQ = 2;
8      GENRESP = 3;
9      // TODO this is just used for the old MMS implementation
10     SERVER = 99;
11 }
12
13 enum ReqResKind {
14     UNSET = -1;
15     READ = 0;
16     COMMAND = 1;
17 }
18
19 enum MITMKind {
20     UNMOD = 0;
21     COMPR = 1;
22     BLOCK = 2;
23     FAKEGEN = 3;
24 }
25
26 class MmsMessage extends FieldsChunk
27 {
28     // Id to keep track of the request associated with a
           certain response (just used in the MITM attack)
29     msgid_t originId;
30     B expectedReplyLength;
31     bool serverClose;
32     MMSKind messageKind;
33     // Just set if the messageKind is set o GENREQ or
           GENRESP
34     ReqResKind reqResKind;
35     // Mantained for backward compatibility with the old
           implementation
36     int connId;
37     // Used just for the MMS MIMT Attack
38     int evilServerConnId = 0;
```

```
39      int serverIndex = 0;
40      MITMKind atkStatus = MITMKind::UNMOD;
41      // Field that defines the message data (0-9)
42      int data = 0;
43  }
```

Codice 5.1: Il file `MmsMessage.msg`

5.2.2 Il client MMS

Come abbiamo già accennato precedentemente l'`MmsClient` rappresenta una versione modificata della `TcpBasicClientApp`. In questo caso abbiamo scelto d'importare direttamente parametri e metodi di quest'ultima (invece di estenderla come fatto nel `ClientEvilComp` dell'attaccante che vedremo in sezione 5.2.5) per meglio comprendere e studiare come la nostra implementazione di MMS si possa integrare nella struttura di base offerta da *INET*.

MmsClient.ned

Possiamo concentrarci inizialmente sul file **`MmsClient.ned`** di cui vedremo esclusivamente i parametri, i segnali e le misure aggiunte come integrazione al `TcpBasicClientApp`, visto che le altre le abbiamo già osservate in sezione 4.2.2. Il primo parametro `isLogging` definisce se il client debba loggare su file i messaggi che invia o riceve, mentre `sentReadInterval` e `sendCommandInterval` rappresentano rispettivamente il tempo tra richieste di lettura di misure e invio di comandi consecutivi. Per ultimo `resTimeoutInterval` è il ritardo con cui dichiarare una risposta come non ricevuta, calcolato a partire dall'invio della richiesta. Nella parte successiva di codice sono dichiarati una serie di segnali e misure per tracciare o in forma di conteggio o vettoriale delle statistiche relative a richieste e risposte di misure e comandi (`readSent`, `commandSent`, `readResponse` e `commandResponse`), comprese quelle inerenti ai rispettivi timeout (`readResponseTimeout` e `commandResponseTimeout`).

```
1   simple MmsClient like IApp {
2       parameters:
3           ...
4
5           // specify if the specific client must log the
                packets on file
6           bool isLogging = default(false);
7           // specify the interval between a read request and
                the next
8           volatile int sendReadInterval @unit(s) = default(
                intuniform(8s, 10s));
9           // specify the interval between a command request
                and the next
10          volatile int sendCommandInterval @unit(s) = default(
                intuniform(12s, 15s));
11          // timeout for generic responses
```

```
12          volatile int resTimeoutInterval @unit(s) = default(1
               s);
13
14          ...
15
16          @signal[measureReceivedCount](type=int);
17          @statistic[measureReceivedAmount](source=
               measureReceivedCount; record=vector);
18
19          @signal[readSentSignal](type=bool);
20          @statistic[readSent](source=readSentSignal; record=
               count);
21
22          @signal[commandSentSignal](type=bool);
23          @statistic[commandSent](source=commandSentSignal;
               record=count);
24
25          @signal[readResponseSignal](type=bool);
26          @statistic[readResponse](source=readResponseSignal;
               record=count);
27
28          @signal[commandResponseSignal](type=bool);
29          @statistic[commandResponse](source=
               commandResponseSignal; record=count);
30
31          @signal[readResponseTimeoutSignal](type=bool);
32          @statistic[readResponseTimeout](source=
               readResponseTimeoutSignal; record=count);
33
34          @signal[commandResponseTimeoutSignal](type=bool);
35          @statistic[commandResponseTimeout](source=
               commandResponseTimeoutSignal; record=count);
36
37          @signal[readResponseReceivedTimeSignal](type=int);
38          @statistic[readResponseReceivedTime](source=
               readResponseReceivedTimeSignal; record=vector);
39
40          @signal[commandResponseReceivedTimeSignal](type=int)
               ;
41          @statistic[commandResponseReceivedTime](source=
               commandResponseReceivedTimeSignal; record=vector)
               ;
42      gates:
43          input socketIn @labels(TcpCommand/up);
44          output socketOut @labels(TcpCommand/down);
45  }
```

Codice 5.2: File `MmsClient.ned`

MmsClient.h

Guardando anche al file d'intestazione **MmsClient.h** presente nel codice 5.3 possiamo trovare immediatamente due tempi d'inizio e fine di operatività del modulo (`startTime` e `stopTime`), una coda di decodifica dei messaggi (`queue`) e un `logger` di pacchetti. In seguito è presente un messaggio associato al timer di timeout dei messaggi (`timeoutMsg`), il relativo tempo di attesa prima di dichiarare una risposta in timeout e una serie di `HashMap`. Queste in particolare solo usate per tenere traccia di ogni self-message di timeout (`cMessage*`) con l'id relativo alla richiesta inviata (`msgid_t`) così che sia possibile risalire all'evento di timeout schedulato per ciascuna richiesta di lettura di misura (`readResTimeoutMap`) o invio di comando (`commandResTimeoutMap`). La `genReqSentTimeMap` serve invece a memorizzare il tempo d'invio di ciascuna richiesta così da poter calcolare il ritardo di ricezione della risposta (particolarmente utile per lo studio di attacchi DDoS). Seguono poi una serie di variabili usate per tracciare il numero di misure (`measureCounter`) ricevute ogni `measureAmountEventDelay`, per cui viene programmato un evento periodico (`measureAmountEvent`) con tale ritardo e viene generato un segnale per propagare il valore calcolato a ogni intervallo (`measureReceivedCount`). Le ultime variabili dichiarate servono a memorizzare se il client è già iscritto per la ricezione di misure periodiche dal server (`isListening`), se sta registrando i pacchetti (`isLogging`) e anche i segnali omonimi già presentati nel file NED. Tra i metodi di cui troviamo il prototipo ne abbiamo uno adibito all'invio di richieste (`sendRquest`), uno per programmare nuovi invii (`rescheduleOrDeleteTimer`), un altro per svolgere l'inizializzazione del modulo `initialize` e un ultimo per la gestione dei self-messages inerenti ai timer. In seguito sono dichiarate una serie di funzioni il monitoraggio delle diverse fasi del socket TCP tra cui la connessione al server (`socketEstablished`), l'arrivo di nuovi dati (`socketDataArrived`), la sua chiusura (`socketClosed`) e il verificarsi di errori sulla connessione (`socketFailure`). Infine altre funzioni gestiscono il ciclo di operatività del modulo tra cui operazioni di avvio (`handleStartOperation`), di fine (`handleStopOperation`) e di riavvio (`handleCrashOperation`).

```
1  class INET_API MmsClient : public TcpAppBase {
2    protected:
3      ...
4      simtime_t startTime;
5      simtime_t stopTime;
6      ChunkQueue queue;
7      MmsPacketLogger* logger = nullptr;
8
9      cMessage *timeoutMsg = nullptr;
10     int resTimeout;
11     std::map<msgid_t, cMessage*> readResTimeoutMap;
12     std::map<msgid_t, cMessage*> commandResTimeoutMap;
13     std::map<msgid_t, simtime_t> genReqSentTimeMap;
14
15
16     //Packet count
17     int measureCounter;
```

```
18      int measureAmountEventDelay = 4;
19      cMessage* measureAmountEvent;
20      simsignal_t measureReceivedCount;
21
22      //Request-response management
23      bool isListening;
24      // Variable to know if the client is logging
25      bool isLogging;
26      simsignal_t readSentSignal;
27      simsignal_t commandSentSignal;
28      simsignal_t readResponseSignal;
29      simsignal_t commandResponseSignal;
30      simsignal_t readResponseTimeoutSignal;
31      simsignal_t commandResponseTimeoutSignal;
32      simsignal_t readResponseReceivedTimeSignal;
33      simsignal_t commandResponseReceivedTimeSignal;
34
35      virtual void sendRequest(MMSKind kind = MMSKind::CONNECT
            , ReqResKind reqKind = ReqResKind::READ);
36      virtual void rescheduleOrDeleteTimer(simtime_t d, short
            int msgKind);
37
38      virtual int numInitStages() const override { return
            NUM_INIT_STAGES; }
39      virtual void initialize(int stage) override;
40      virtual void handleTimer(cMessage *msg) override;
41
42      virtual void socketEstablished(TcpSocket *socket)
            override;
43      virtual void socketDataArrived(TcpSocket *socket, Packet
             *msg, bool urgent) override;
44      virtual void socketClosed(TcpSocket *socket) override;
45      virtual void socketFailure(TcpSocket *socket, int code)
            override;
46
47      virtual void handleStartOperation(LifecycleOperation *
            operation) override;
48      virtual void handleStopOperation(LifecycleOperation *
            operation) override;
49      virtual void handleCrashOperation(LifecycleOperation *
            operation) override;
50
51      virtual void close() override;
52
53      ...
54  };
```

Codice 5.3: File `MmsClient.h`

MmsClient.cc

È nel file **`MmsClient.cc`** (visibile nel codice 5.4) che troviamo le definizioni dei corpi dei metodi visti nel rispettivo file d'intestazione. Partendo dal metodo **`initialize`**, vengono inizializzati parametri come `startTime`, `stopTime`, `resTimeout` e il `logger` per la registrazione di messaggi inviati e ricevuti. Nelle righe che seguono vengono registrati i segnali dichiarati nel file NED e dopo aver azzerato il contatore delle misure, viene programmato il primo self-message per il monitoraggio del numero di messaggi periodici ricevuti nell'intervallo temporale prestabilito. La funzione **`socketEstablished`** viene chiamata nel momento in cui la connessione TCP è stabilita con successo e in questa occasione viene subito inviato al server MMS il messaggio d'iscrizione del client per la ricezione di misure periodiche. Immediatamente dopo vengono anche schedulati gli eventi d'invio di una richiesta di lettura di variabile (in base al delay `sendReadInterval`) e d'invio di comando (secondo il delay `sendCommandInterval`). Questa operazione è svolta all'interno del metodo **`rescheduleOrDeleteTimer`** che programma l'invio di un messaggio del tipo indicato e con il ritardo passato come parametro (sempre che il tempo di simulazione risultante non superi l'istante di `stopTime`). Nel metodo **`handleTimer`** vengono eseguite le azioni associate ai diversi tipi di self-messages che il client può programmare. Se il messaggio scattato è un `measureAmountEvent` allora viene emesso il segnale relativo al numero di misure ricevute nell'ultimo intervallo di tempo e viene schedulato nuovamente tale evento. Nelle altre ipotesi il messaggio viene gestito sulla base della sua tipologia che può essere:

- `MSGKIND_CONNECT`: Viene inizializzata la connessione con il server MMS.
- `MSGKIND_SEND_READ`: Viene inviata una nuova richiesta di lettura di misura e un nuovo timer d'invio viene programmato in base al delay ottenuto da `sendReadInterval` all'interno del metodo `rescheduleOrDeleteTimer`.
- `MSGKIND_SEND_COMMAND`: Viene inviata una nuova richiesta di esecuzione di comando e un nuovo timer d'invio viene programmato in base al delay ottenuto da `sendCommandInterval` all'interno del metodo `rescheduleOrDeleteTimer`.
- `MSGKIND_RES_TIMEOUT`: Rappresenta lo scatto di un timer associato a un evento di timeout di una richiesta inviata. Questo implica che per prima cosa si cerchi il messaggio scattato nella HashMap che traccia i timeout di messaggi di lettura (`readResTimeoutMap`) e nel caso in cui venga trovato viene eliminato e viene emesso il segnale corrispondente. Se lì non è presente allora si compie la stessa ricerca nella struttura dati corrispondente per le richieste di esecuzione di comandi (`commandResTimeoutMap`) e in caso positivo viene eliminato e viene emesso il segnale associato.

Nel metodo **`sendRequest`** troviamo tutte quelle azioni che vengono compiute quando viene richiesto l'invio di un messaggio MMS. Dopo aver creato un puntatore condiviso (`payload = makeShared<MmsMessage>()`) si vanno a inizializzare `originId` (per associare richieste e risposte), la `chunkLength` (dimensione della richiesta), la `expectedReplyLength` (dimensione attesa della risposta),

`creationTime` (tempo di generazione del messaggio), il `serverIndex` (indice del server con cui il client vuole comunicare che dipende dall'indice dell'applicazione client nell'host), il campo `data` (al momento sempre a 0) e l'`atkStatus` (inizialmente configurato come non alterato). Se il client non è ancora iscritto per la ricezione di misure periodiche allora viene inviato un messaggio di tipo `CONNECT`, altrimenti in base al fatto che la richiesta sia di tipo `READ` o `COMMAND` viene inserito e programmato un nuovo evento di timeout nella `readResTimeoutMap` o nella `commandResTimeoutMap` e viene emesso un segnale d'invio di messaggio (oltre che memorizzato il tempo di simulazione corrente nella `genReqSentTimeMap`). Il messaggio MMS viene poi incapsulato in un pacchetto da inviare con il metodo `sendPacket` e viene registrato su file se necessario. L'ultima funzione che vedremo di questa classe è la **`socketDataArrived`** che gestisce i messaggi in arrivo sul socket TCP stabilito con il server MMS. Dopo aver verificato se la connessione è stata chiusa, il messaggio MMS viene estratto dal pacchetto e decodificato. Se abbiamo ricevuto una misura periodica allora aggiorniamo il contatore, mentre se ci è arrivata una risposta di lettura di variabile o un ACK d'esecuzione di comando allora viene emesso un segnale di ricezione di messaggio con anche il tempo impiegato (l'istante d'invio era stato memorizzato in `genReqSentTimeMap`). Inoltre bisogna andare a rimuovere dalla corrispondente HashMap e anche dalla FEL l'evento di timeout associato alla richiesta (operazione fattibile grazie all'`originId`), così che non generi un falso allarme.

```
1  Define_Module(MmsClient);
2
3  ...
4
5  void MmsClient::initialize(int stage)
6  {
7      TcpAppBase::initialize(stage);
8      if (stage == INITSTAGE_LOCAL) {
9          ...
10         cEnvir* ev = getSimulation()->getActiveEnvir();
11         startTime = par("startTime");
12         stopTime = par("stopTime");
13         resTimeout = par("resTimeoutInterval");
14         isLogging = par("isLogging");
15         if(isLogging) {
16           logger = new MmsPacketLogger(ev->getConfigEx()->
               getActiveRunNumber(), "client", getParentModule
               ()->getIndex(), getIndex());
17         }
18         if (stopTime >= SIMTIME_ZERO && stopTime < startTime
               )
19             throw cRuntimeError("Invalid startTime/stopTime
                   parameters");
20         timeoutMsg = new cMessage("timer");
21
22         measureAmountEvent = new cMessage("Topic Amount
```

```
            Event");
23          measureReceivedCount = registerSignal("
               measureReceivedCount");
24          readSentSignal = registerSignal("readSentSignal");
25          commandSentSignal = registerSignal("
               commandSentSignal");
26          readResponseSignal = registerSignal("
               readResponseSignal");
27          commandResponseSignal = registerSignal("
               commandResponseSignal");
28          readResponseTimeoutSignal = registerSignal("
               readResponseTimeoutSignal");
29          commandResponseTimeoutSignal = registerSignal("
               commandResponseTimeoutSignal");
30          readResponseReceivedTimeSignal = registerSignal("
               readResponseReceivedTimeSignal");
31          commandResponseReceivedTimeSignal = registerSignal("
               commandResponseReceivedTimeSignal");
32
33
34          measureCounter = 0;
35          isListening = false;
36          scheduleAt(simTime() + SimTime(
               measureAmountEventDelay, SIMTIME_S),
               measureAmountEvent);
37      }
38  }
39
40  ...
41
42  void MmsClient::socketEstablished(TcpSocket *socket)
43  {
44      TcpAppBase::socketEstablished(socket);
45
46      ...
47
48      // Send the register for measure MMS message
49      sendRequest();
50
51      ...
52
53      // Schedule a read send
54      simtime_t dRead = simTime() + SimTime(par("
           sendReadInterval").intValue(), SIMTIME_S);
55      rescheduleOrDeleteTimer(dRead, MSGKIND_SEND_READ);
56      // // Schedule a command send
57      simtime_t dCommand = simTime() + SimTime(par("
           sendCommandInterval").intValue(), SIMTIME_S);
```

```
58      rescheduleOrDeleteTimer(dCommand, MSGKIND_SEND_COMMAND);
59  }
60
61  void MmsClient::rescheduleOrDeleteTimer(simtime_t d, short
        int msgKind)
62  {
63      //cancelEvent(timeoutMsg);
64    timeoutMsg = new cMessage("timer");
65
66      if (stopTime < SIMTIME_ZERO || d < stopTime) {
67          timeoutMsg->setKind(msgKind);
68          scheduleAt(d, timeoutMsg);
69      }
70      else {
71          delete timeoutMsg;
72          timeoutMsg = nullptr;
73      }
74  }
75
76  void MmsClient::handleTimer(cMessage *msg)
77  {
78      if (msg == measureAmountEvent) {
79          emit(measureReceivedCount, measureCounter);
80          measureCounter = 0;
81          scheduleAt(simTime() + SimTime(
                measureAmountEventDelay, SIMTIME_S),
                measureAmountEvent);
82          return;
83      }
84      simtime_t dRead = 0;
85      simtime_t dCommand = 0;
86      bool found = false;
87      switch (msg->getKind()) {
88          case MSGKIND_CONNECT:
89              connect();
90              delete msg;
91              break;
92
93          case MSGKIND_SEND_READ:
94              sendRequest(MMSKind::GENREQ, ReqResKind::READ);
95              numRequestsToSend--;
96              delete msg;
97              // Schedule a Read send
98              dRead = simTime() + SimTime(par("
                    sendReadInterval").intValue(), SIMTIME_S);
99              rescheduleOrDeleteTimer(dRead, MSGKIND_SEND_READ
                    );
100             break;
```

```
101
102         case MSGKIND_SEND_COMMAND:
103             sendRequest(MMSKind::GENREQ, ReqResKind::COMMAND
                    );
104             numRequestsToSend--;
105             delete msg;
106             // Schedule a Command send
107             dCommand = simTime() + SimTime(par("
                    sendCommandInterval").intValue(), SIMTIME_S);
108             rescheduleOrDeleteTimer(dCommand,
                    MSGKIND_SEND_COMMAND);
109             break;
110
111         case MSGKIND_RES_TIMEOUT:
112             for(auto &i : readResTimeoutMap) {
113                 if (i.second == msg) {
114                     readResTimeoutMap.erase(i.first);
115                     genReqSentTimeMap.erase(i.first);
116                     delete msg;
117                     // Emit signal for generic response
                            timeout
118                     emit(readResponseTimeoutSignal, true);
119                     found = true;
120                     break;
121                 }
122             }
123             if(!found) {
124                 for(auto &i : commandResTimeoutMap) {
125                     if (i.second == msg) {
126                         commandResTimeoutMap.erase(i.first);
127                         genReqSentTimeMap.erase(i.first);
128                         delete msg;
129                         // Emit signal for generic response
                                timeout
130                         emit(commandResponseTimeoutSignal,
                                true);
131                         break;
132                     }
133                 }
134             }
135
136             break;
137
138         default:
139             throw cRuntimeError("Invalid timer msg: kind=%d"
                    , msg->getKind());
140     }
141 }
```

```
142
143 void MmsClient::sendRequest(MMSKind kind, ReqResKind reqKind
    ) {
144     ...
145
146     const auto& payload = makeShared<MmsMessage>();
147     Packet *packet = new Packet("data");
148     payload->setOriginId(packet->getId());
149     payload->setChunkLength(B(requestLength));
150     payload->setExpectedReplyLength(B(replyLength));
151     payload->setServerClose(false);
152     payload->addTag<CreationTimeTag>()->setCreationTime(
        simTime());
153     payload->setServerIndex(this->getIndex());
154     payload->setData(0);
155     payload->setAtkStatus(MITMKind::UNMOD);
156     EV << "Index: " << this->getIndex();
157     if(!isListening && kind == MMSKind::CONNECT) {
158       // Connect kind
159         payload->setMessageKind(kind);
160         isListening = true;
161     }
162     else {
163         payload->setMessageKind(kind);
164         // Send a Read or a Command
165         payload->setReqResKind(reqKind);
166
167         // Add the timeout event for the respective response
168     cMessage* resTimeoutMsg = new cMessage("Response timeout
        ");
169     resTimeoutMsg->setKind(MSGKIND_RES_TIMEOUT);
170
171         if(reqKind == ReqResKind::READ) {
172           readResTimeoutMap.insert({payload->getOriginId(),
              resTimeoutMsg});
173           emit(readSentSignal, true);
174         } else if(reqKind == ReqResKind::COMMAND) {
175           commandResTimeoutMap.insert({payload->getOriginId
              (), resTimeoutMsg});
176           emit(commandSentSignal, true);
177         }
178
179         scheduleAt(simTime() + SimTime(resTimeout, SIMTIME_S
            ), resTimeoutMsg);
180
181         // Memorize when the message has been sent
182         genReqSentTimeMap.insert({payload->getOriginId(),
            simTime()});
```

```
183     }
184
185     packet->insertAtBack(payload);
186
187     if(isLogging) logger->log(payload.get(), simTime());
188
189     sendPacket(packet);
190 }
191
192 void MmsClient::socketDataArrived(TcpSocket *socket, Packet
        *msg, bool urgent)
193 {
194     if (socket->getState() == TcpSocket::LOCALLY_CLOSED) {
195         EV_INFO << "reply to last request arrived, closing
                session\n";
196         close();
197         return;
198     }
199     auto chunk = msg->peekDataAt(B(0), msg->getTotalLength()
            );
200     queue.push(chunk);
201     while (const auto& appmsg = queue.pop<MmsMessage>(b(-1),
            Chunk::PF_ALLOW_NULLPTR)) {
202       if(isLogging) logger->log(const_cast<MmsMessage*>(
             appmsg.get()), simTime());
203         if(appmsg->getMessageKind() == MMSKind::MEASURE) {
204           measureCounter++;
205         }
206         if(appmsg->getMessageKind() == MMSKind::GENRESP) {
207           simtime_t sendTime = genReqSentTimeMap.find(appmsg
                ->getOriginId())->second;
208           if(appmsg->getReqResKind() == ReqResKind::READ) {
209               emit(readResponseSignal, true);
210               emit(readResponseReceivedTimeSignal, simTime()
                    - sendTime);
211               if(readResTimeoutMap.find(appmsg->getOriginId
                    ()) != readResTimeoutMap.end()) {
212                 cMessage* tmpTimeout = readResTimeoutMap[
                      appmsg->getOriginId()];
213                 readResTimeoutMap.erase(appmsg->getOriginId
                      ());
214                 cancelAndDelete(tmpTimeout);
215               }
216           } else if(appmsg->getReqResKind() == ReqResKind::
                COMMAND) {
217               emit(commandResponseSignal, true);
218               emit(commandResponseReceivedTimeSignal,
                    simTime() - sendTime);
```

```
219                 if(commandResTimeoutMap.find(appmsg->
                        getOriginId()) != commandResTimeoutMap.end
                        ()) {
220                   cMessage* tmpTimeout = commandResTimeoutMap[
                          appmsg->getOriginId()];
221                   commandResTimeoutMap.erase(appmsg->
                          getOriginId());
222                   cancelAndDelete(tmpTimeout);
223                 }
224             }
225             genReqSentTimeMap.erase(appmsg->getOriginId());
226           }
227       }
228       TcpAppBase::socketDataArrived(socket, msg, urgent);
229   }
230
231   ...
```

Codice 5.4: File `MmsClient.cc`

5.2.3 Il server MMS

Anche l'`MmsServer` rappresenta una versione arricchita della `TcpGenericServerApp` e come per il client abbiamo scelto d'importare direttamente la sua implementazione invece di estenderla (come fatto nel `ServerEvilComp` dell'attaccante in sezione 5.2.5) così da poterne studiare meglio le scelte.

MmsServer.ned

Iniziando anche in questo caso dal file **`MmsServer.ned`** (visibile nel codice 5.5), non presenteremo tutti i parametri già visti per la `TcpGenericServerApp` in sezione 4.2.2, ma solo quelli più importanti introdotti o usati per le funzionalità principali del `MmsServer`. Successivamente al parametro di configurazione del logging, troviamo un `replyDelay` impostabile come ritardo aggiuntivo alla risposta dei pacchetti ricevuti, oltre che un `serviceTime` introdotto come tempo di servizio vero e proprio per la gestione dei pacchetti MMS. Infine in `emitInterval` abbiamo il delay per l'invio delle misure periodiche ai client iscritti.

```
1   simple MmsServer like IApp {
2       parameters:
3           ...
4
5           // Parameter to know if the server has to log all
                the packets it receives
6           bool isLogging = default(false);
7           // Additional delay on replies
8           volatile int replyDelay @unit(ms) = default(0ms);
9           // Time to manage a new incoming message
```

```
10          volatile int serviceTime @unit(ms) = default(
                intuniform(2ms, 4ms));
11          volatile int emitInterval @unit(ms) = default(4s);
12      gates:
13          input socketIn @labels(TcpCommand/up);
14          output socketOut @labels(TcpCommand/down);
15  }
```

Codice 5.5: File `MmsServer.ned`

MmsServer.h

Passando al file d'intestazione **`MmsServer.h`** (visibile nel codice 5.6) possiamo vedere in seguito alle dichiarazioni di estensione delle classi, una variabile `socket` per rimanere in ascolto di tutte le comunicazioni e dei messaggi in entrata. Al contrario la HashMap `socketQueue` rappresenta quelli che sono i canali TCP d'uscita verso i singoli client connessi e permette di far tornare indietro i messaggi a loro indirizzati. Invece la lista di coppie `clientConnIdList` memorizza tutti i client iscritti per la ricezione di misure periodiche e in particolare tiene traccia oltre dell'id di connessione primario, anche dell'id di connessione di un attaccante intermedio verso il client compromesso. Se da una parte questa caratteristica risulta completamente irrealistica, dall'altra rappresenta l'unico modo per riuscire a simulare tale scenario pur potendo agire solo a livello applicativo. In ogni caso l'aggiunta di questo campo alle misure periodiche non va a influenzare le statistiche ottenute al termine della simulazione. In seguito una serie di variabili tra cui `serverStatus`, `serverQueue` e `departureEvent` garantiscono la corretta operatività del server in fase dei risposta alla richiese MMS ricevute. Il self-message `sendDataEvent` scandisce l'invio di misure periodiche, mentre `isLogging` e `logger` permettono al server di registrare su file i messaggi inviati e ricevuti. Tra i vari metodi del server troviamo `sendBack` (si occupa dell'invio vero e proprio dei messaggi), `sendOrSchedule` (richiama il metodo `sendBack` o ritarda ulteriormente l'invio se il `replyDelay` è maggiore di 0), `initialize` (svolge i compiti d'inizializzazione del modulo), `handleMessage` (gesisce sia i self-messages che i messaggi giunti in entrata dai client), `handleDeparture` (programma i messaggi in uscita), `sendPacketDeparture` (compone i messaggi MMS in uscita) e `logPacket` (registra su file i pacchetti transitati).

```
1  class INET_API MmsServer : public cSimpleModule, public
       LifecycleUnsupported {
2    protected:
3      TcpSocket socket;
4      ...
5      std::map<int, ChunkQueue> socketQueue;
6      std::list< std::pair<int,int> > clientConnIdList;
7
8      // Queue and server management
9      bool serverStatus;
10     cQueue serverQueue;
11     cMessage* departureEvent;
```

```
12
13      // Measures send management
14      cMessage* sendDataEvent;
15
16      // Packet logging
17      bool isLogging;
18      MmsPacketLogger* logger;
19
20    protected:
21      virtual void sendBack(cMessage *msg);
22      virtual void sendOrSchedule(cMessage *msg, simtime_t
           delay);
23
24      virtual void initialize(int stage) override;
25      ...
26      virtual void handleMessage(cMessage *msg) override;
27      ...
28      virtual void handleDeparture();
29      virtual void sendPacketDeparture(int connId, msgid_t
           originId, int evilConnId, B requestedBytes, B
           replyLength,
30           MMSKind messageKind, ReqResKind reqResKind, int data
              , MITMKind atkStatus);
31
32      void logPacket(Packet* packet);
33
34    public:
35      virtual ~MmsServer();
36  };
```

Codice 5.6: File `MmsServer.h`

MmsServer.cc

Concludiamo l'analisi dell'`MmsServer` dando uno sguardo al file **`MmsServer.cc`**, visibile nel codice 5.7. All'interno del metodo **`intialize`**, in seguito all'inizializzazione di tutto quello che concerne il funzionamento del socket TCP (bind su `localAddress` e `localPort`) e dopo averlo messo in modalità di ascolto, vengono inizializzati i self-messages usati dal server (`departureEvent` e `sendDataEvent`). Infine dopo aver programmato l'invio della prima misura periodica ad 1 secondo dall'inizio della simulazione, viene inizializzato l'oggetto necessario alle operazioni di logging su file e viene impostato ad `UP` lo stato di operatività del server.
Il metodo **`handleMessage`** viene invece chiamato da *OMNeT++* quando è necessario gestire un messaggio o un evento indirizzato o programmato per il server MMS. Nel caso in cui il messaggio estratto dalla FEL sia il `departureEvent` viene gestito l'invio di un nuovo messaggio ai client tramite il metodo `handleDeparture`, mentre se è un `sendDataEvent` allora viene creato un nuovo pacchetto contenente una misura periodica (`messageKind` impostato a `MEASURE` e `chunkLength` di

100bytes) e se la `clientConnIdList` non è vuota, tale messaggio viene inserito nella `serverQueue` e il suo invio programmato in base al `serviceTime` configurato (se il server non è già impegnato in un altra comunicazione). Se nessuna delle condizioni precedenti è verificata allora ci troviamo nella situazione di dover gestire un pacchetto giunto dal socket TCP, le cui tipologie (`msg->getKind()`) e i conseguenti comportamenti possono differire:

- `TCP_I_PEER_CLOSED`: Un client connesso ha scelto di chiudere la connessione TCP, ma tale operazione verrà svolta solo quando non vi saranno più messaggi in attesa di essere spediti su quella connessione, così scheduliamo una richiesta di chiusura che impartirà la terminazione vera e propria della connessione solo dopo l'invio dei messaggi che la precedono.

- `TCP_I_DATA` o `TCP_I_URGENT_DATA`: È stato ricevuto un messaggio da parte di un client, di conseguenza lo inseriamo nella `serverQueue` e programmiamo la sua gestione con ritardo `serviceTime` (se il server è libero).

- `TCP_I_AVAILABLE`: Il socket notifica la sua operatività così che il livello applicativo memorizzi di poterne fare uso.

Se il messaggio ricevuto non ricade in nessuna delle casistiche precedenti allora viene scartato con un avvertimento di errore.
Come già accennato quando è necessario gestire l'invio di un nuovo messaggio viene effettuata una chiamata al metodo **`handleDeparture`**. Siccome il prossimo messaggio a cui andremo a rispondere è contenuto nella `serverQueue`, per prima cosa lo si va a estrarre e dopo averne segnalato la ricezione (`packetReceivedSignal`) si procede con la decodifica nel ciclo `while`. A questo punto il metodo prevede diverse casistiche in base al tipo di messaggio ottenuto (`appmsg->getMessageKind()`):

- `MMSKind::CONNECT`: Stiamo gestendo un messaggio di registrazione da parte di un client quindi memorizziamo il suo `connId` ed `evilServerConnId` nella `clientConnIdList`.

- `MMSKind::MEASURE`: Stiamo gestendo l'invio periodico di misure ai client iscritti quindi per ogni elemento della `clientConnIdList` creiamo un nuovo messaggio di misura con le caratteristiche di quello appena estratto e lo inviamo al client individuato dall'id di connessione.

- `MMSKind::GENREQ`: Stiamo gestendo un messaggio di richiesta di lettura di misura da parte di un client, quindi creiamo un nuovo messaggio di risposta (`GENRESP`) con parte dei parametri presi dalla richiesta (`originId`, `evilServerConnId`, `expectedReplyLength` come dimensione del pacchetto e stesso `reqResKind`).

Se inoltre il parametro `serverClose` nel messaggio è stato impostato a `true` allora il corrispondente socket TCP con il client viene chiuso e quest'ultimo viene anche rimosso dalla `clientConnIdList`. Al termine della funzione, se la `serverQueue` non è vuota, il self-message `departureEvent` viene riprogrammato secondo il `serviceTime` determinato.

Già nel metodo immediatamente precedente abbiamo potuto trovare diverse chiamate della funzione **sendPacketDeparture**. Il suo scopo è quello di generare un nuovo messaggio MMS all'interno di un puntatore condiviso tra moduli, utilizzando i parametri passati alla funzione. Questo messaggio MMS viene poi incapsulato in un pacchetto e inviato attraverso il metodo `sendOrSchedule` (già citato durante l'analisi del file d'intestazione).
Infine la funzione **logPacket** prende il pacchetto passato come parametro, usa una `ChunkQueue` temporanea per decodificarlo in un messaggio MMS e lo registra su file attraverso il `logger`.

```
1  Define_Module(MmsServer);
2
3  void MmsServer::initialize(int stage) {
4      cSimpleModule::initialize(stage);
5
6      if (stage == INITSTAGE_LOCAL) {
7          ...
8      }
9      else if (stage == INITSTAGE_APPLICATION_LAYER) {
10         const char *localAddress = par("localAddress");
11         int localPort = par("localPort");
12         socket.setOutputGate(gate("socketOut"));
13         socket.bind(localAddress[0] ? L3AddressResolver().
               resolve(localAddress) : L3Address(), localPort);
14         socket.listen();
15         departureEvent = new cMessage("Server Departure");
16         sendDataEvent = new cMessage("Register Data To Send
               Event");
17         serverStatus = false;
18         scheduleAt(1, sendDataEvent);
19
20         cEnvir* ev = getSimulation()->getActiveEnvir();
21         isLogging = par("isLogging").boolValue();
22         if(isLogging) {
23           logger = new MmsPacketLogger(ev->getConfigEx()->
                 getActiveRunNumber(), "server", getParentModule
                 ()->getIndex(), getIndex());
24         }
25
26         cModule *node = findContainingNode(this);
27         NodeStatus *nodeStatus = node ?
               check_and_cast_nullable<NodeStatus *>(node->
               getSubmodule("status")) : nullptr;
28         bool isOperational = (!nodeStatus) || nodeStatus->
               getState() == NodeStatus::UP;
29         if (!isOperational)
30             throw cRuntimeError("This module doesn't support
                   starting in node DOWN state");
```

```
31      }
32  }
33
34  ...
35
36  void MmsServer::handleMessage(cMessage *msg) {
37      if (msg == departureEvent) handleDeparture();
38
39      else if(msg == sendDataEvent) {
40          auto pkt = new Packet("SendData");
41          pkt->addTag<SocketInd>()->setSocketId(-1);
42          const auto& payload = makeShared<MmsMessage>();
43          payload->setMessageKind(MMSKind::MEASURE);
44          payload->setChunkLength(B(100));
45          payload->setExpectedReplyLength(B(100));
46          payload->setServerClose(false);
47          payload->setData(0);
48          payload->setAtkStatus(MITMKind::UNMOD);
49          pkt->insertAtBack(payload);
50          if (!clientConnIdList.empty()) {
51              if(!serverStatus) {
52                  serverStatus = true;
53                  serverQueue.insert(pkt);
54                  scheduleAt(simTime() + SimTime(par("
                        serviceTime").intValue(), SIMTIME_MS),
55                          departureEvent);
56              }
57              else serverQueue.insert(pkt);
58          }
59          scheduleAt(simTime() + SimTime(par("emitInterval").
                intValue(), SIMTIME_MS), sendDataEvent);
60      }
61
62      else if (msg->isSelfMessage()) {
63          sendBack(msg);
64      }
65      else if (msg->getKind() == TCP_I_PEER_CLOSED) {
66
67          // we'll close too, but only after there's surely no
                message
68          // pending to be sent back in this connection
69          int connId = check_and_cast<Indication *>(msg)->
                getTag<SocketInd>()->getSocketId();
70          delete msg;
71          auto request = new Request("close", TCP_C_CLOSE);
72          request->addTag<SocketReq>()->setSocketId(connId);
73          clientConnIdList.remove_if([&](std::pair<int, int>&
                p) {
```

```
74            return p.first == connId;
75          });
76          sendOrSchedule(request, SimTime(par("replyDelay").
               intValue(), SIMTIME_MS));
77      }
78      else if (msg->getKind() == TCP_I_DATA || msg->getKind()
           == TCP_I_URGENT_DATA) {
79        if(isLogging) logPacket(check_and_cast<Packet*>(msg));
80          if(!serverStatus) {
81              serverStatus = true;
82              serverQueue.insert(msg);
83              scheduleAt(simTime() + SimTime(par("serviceTime"
                   ).intValue(), SIMTIME_MS),
84                    departureEvent);
85          }
86          else serverQueue.insert(msg);
87      }
88      else if (msg->getKind() == TCP_I_AVAILABLE)
89          socket.processMessage(msg);
90      else {
91          // some indication -- ignore
92          EV_WARN << "drop msg: " << msg->getName() << ", kind
               :" << msg->getKind() << "(" << cEnum::get("inet::
               TcpStatusInd")->getStringFor(msg->getKind()) << "
               )\n";
93          delete msg;
94      }
95  }
96
97  void MmsServer::handleDeparture() {
98      Packet *packet = check_and_cast<Packet *>(serverQueue.
           pop());
99      int connId = packet->getTag<SocketInd>()->getSocketId();
100     ChunkQueue &queue = socketQueue[connId];
101     auto chunk = packet->peekDataAt(B(0), packet->
           getTotalLength());
102     queue.push(chunk);
103     emit(packetReceivedSignal, packet);
104     bool doClose = false;
105     while (queue.has<MmsMessage>(b(-1))) {
106       const auto& appmsg = queue.pop<MmsMessage>(b(-1));
107         msgsRcvd++;
108         bytesRcvd += B(appmsg->getChunkLength()).get();
109         B requestedBytes = appmsg->getExpectedReplyLength();
110         if(appmsg->getMessageKind() == MMSKind::CONNECT) {
               // Register a listener
111             clientConnIdList.push_back({connId, appmsg->
                   getEvilServerConnId()});
```

```
112         }
113         else if(appmsg->getMessageKind() == MMSKind::MEASURE
                ) { // Send data to listeners
114             for (auto const& listener : clientConnIdList) {
115                 if (requestedBytes > B(0)) {
116                     sendPacketDeparture(listener.first,
                        appmsg->getOriginId(), listener.
                        second, B(100), B(0),
117                         MMSKind::MEASURE, ReqResKind::UNSET,
                            appmsg->getData(), appmsg->
                            getAtkStatus());
118                 }
119             }
120         }
121         else if (appmsg->getMessageKind() == MMSKind::GENREQ
                ) { // Response to Generic Request
122             if (requestedBytes > B(0)) {
123                 sendPacketDeparture(connId, appmsg->
                    getOriginId(), appmsg->
                    getEvilServerConnId(), requestedBytes, B
                    (0),
124                     MMSKind::GENRESP, appmsg->getReqResKind
                        (), 0, MITMKind::UNMOD);
125             }
126         }
127         else {
128             //Bad Request
129         }
130         if (appmsg->getServerClose()) {
131             doClose = true;
132             break;
133         }
134     }
135     delete packet;
136
137     if (doClose) {
138         auto request = new Request("close", TCP_C_CLOSE);
139         TcpCommand *cmd = new TcpCommand();
140         request->addTag<SocketReq>()->setSocketId(connId);
141         request->setControlInfo(cmd);
142         clientConnIdList.remove_if([&](std::pair<int, int>&
                p) {
143             return p.first == connId;
144         });
145         sendOrSchedule(request, SimTime(par("replyDelay").
            intValue(), SIMTIME_MS));
146     }
147     if(serverQueue.getLength() > 0) {
```

```
148         scheduleAt(simTime() + SimTime(par("serviceTime").
               intValue(), SIMTIME_MS),
149                 departureEvent);
150     } else serverStatus = false;
151 }
152
153 void MmsServer::sendPacketDeparture(int connId, msgid_t
    originId, int evilConnId, B requestedBytes, B replyLength
    ,
154     MMSKind messageKind, ReqResKind reqResKind, int data,
           MITMKind atkStatus) {
155     Packet *outPacket = new Packet("Generic Data",
           TCP_C_SEND);
156     outPacket->addTag<SocketReq>()->setSocketId(connId);
157     const auto& payload = makeShared<MmsMessage>();
158     payload->setOriginId(originId);
159     payload->setMessageKind(messageKind);
160     payload->setReqResKind(reqResKind);
161     payload->setChunkLength(requestedBytes);
162     payload->setExpectedReplyLength(replyLength);
163     payload->addTag<CreationTimeTag>()->setCreationTime(
           simTime());
164     payload->setEvilServerConnId(evilConnId);
165     payload->setData(data);
166     payload->setAtkStatus(atkStatus);
167     outPacket->insertAtBack(payload);
168     sendOrSchedule(outPacket, SimTime(par("replyDelay").
           intValue(), SIMTIME_MS));
169 }
170
171 void MmsServer::logPacket(Packet* packet) {
172   ChunkQueue tmpQueue;
173   auto chunk = packet->peekDataAt(B(0), packet->
        getTotalLength());
174   tmpQueue.push(chunk);
175   while (tmpQueue.has<MmsMessage>(b(-1))) {
176       const auto& appmsg = tmpQueue.pop<MmsMessage>(b(-1));
177       logger->log(const_cast<MmsMessage*>(appmsg.get()),
             simTime());
178   }
179 }
```

Codice 5.7: File `MmsServer.cc`

5.2.4 Il client malevolo (DDoS)

Un client malevolo che simuli il comportamento di una macchina compromessa per portare a termine un attacco di Distributed Denial Of Service non presenta grandi

differenze rispetto all'`MmsClient` visto in precedenza. Dopo aver stabilito infatti la connessione TCP con il server MMS, inizia a generare una serie di richieste MMS malformate a un tasso d'invio ben superiore a quello utilizzato da un normale client. Quello che è bastato fare nella realizzazione del modulo **`BadMmsClient`** è stato importare le funzionalità già presenti nell'`MmsClient` andando a modificare il metodo **`sendRequest`**, come mostrato nel codice 5.8. In particolare a riga 10 il tipo del messaggio MMS viene impostato a `FAKE` per segnalare che quel pacchetto rappresenta una richiesta che il server non saprà riconoscere e gestire, pur tenendone occupate delle risorse.

```
1  void BadMmsClient::sendRequest() {
2      ...
3
4      const auto& payload = makeShared<MmsMessage>();
5      Packet *packet = new Packet("data");
6      payload->setChunkLength(B(requestLength));
7      payload->setExpectedReplyLength(B(replyLength));
8      payload->setServerClose(false);
9      payload->addTag<CreationTimeTag>()->setCreationTime(
           simTime());
10     payload->setMessageKind(MMSKind::FAKE);
11
12     packet->insertAtBack(payload);
13
14     sendPacket(packet);
15
16     ...
17 }
```

Codice 5.8: Estratto del file `BadMmsClient.cc`

A questo punto in fase di configurazione (o impostandolo come valore di default) è sufficiente indicare un `thinkTime` nell'ordine di qualche millisecondo per permettere al client di generare un elevato numero di richieste in poco tempo.

5.2.5 Il client malevolo (MITM)

Mentre per la modellazione di moduli compromessi per l'attacco DDoS siamo riusciti a basarci in larga parte alla strutturazione offerta di default da *INET*, nel simulare un'entità malevola in un attacco di tipo Man In The Middle la situazione si complica. Questo perché un attaccante che compie uno split della connessione *TCP* deve poter ricevere i messaggi provenienti dal client compromesso per poterli bloccare, alterare e trasmettere ai server con cui sta comunicando, ma deve anche poter svolgere l'operazione inversa. Siccome stiamo operando a livello applicativo dobbiamo tradurre questo comportamento in componenti *INET* (le `TCPApplications`) che hanno determinate limitazioni e ciò non è immediato da realizzare. Una **componente client** (`TcpBasicClientApp`) infatti può collegarsi a un solo server in un determinato istante della simulazione e non ha la possibilità di ricevere messaggi da altri client allo stesso tempo. Invece una **componente server** (`TcpGenericServerApp`) può comunicare con più client che si collegano a essa, ma non può collegarsi a un

altro server della rete. Ciò implica che l'attaccante non possa essere formato da una sola di queste applicazioni, ma necessiti di una loro adeguata combinazione.

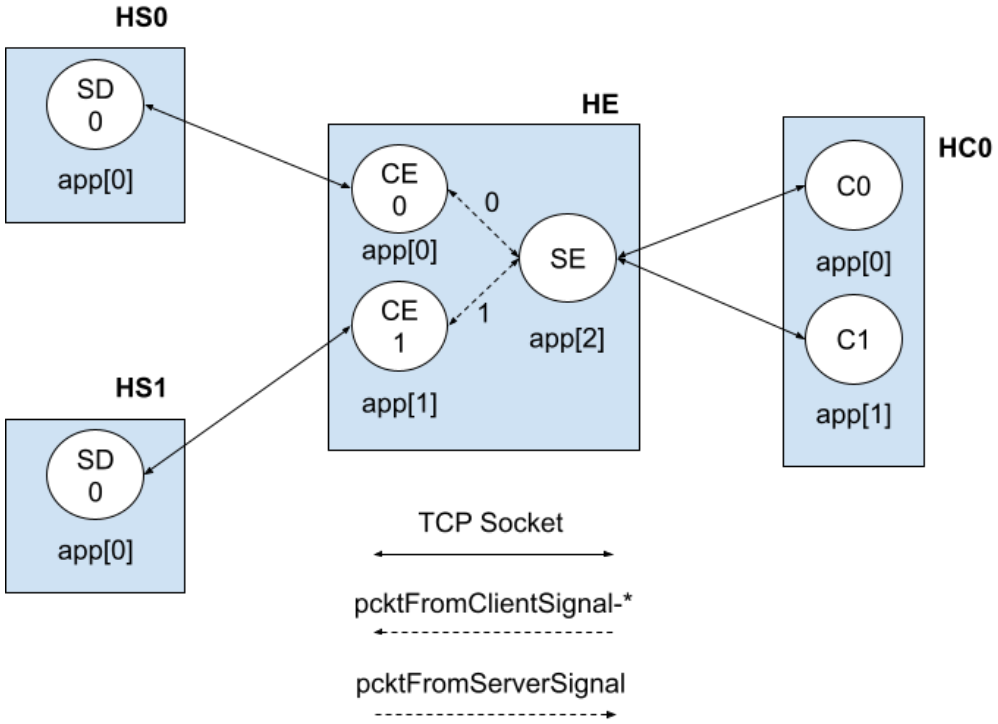


Figura 5.1: Struttura delle comunicazioni nell'attacco MITM

Come possiamo vedere in Figura 5.1 supponiamo di avere una comunicazione che di norma dovrebbe intercorrere tra un host client *HC0* e due host server *HS0* e *HS1*. Se le connessioni non fossero compromesse avremmo che le applicazioni *C0* e *C1* di *HC0* (entrambe di tipo `MmsClient`) dovrebbero collegarsi rispettivamente alle applicazioni *SD0* di *HS0* e *SD0* di *HS1* (entrambe di tipo `MmsServer`). In realtà siccome in questo modello l'attaccante *HE* è riuscito a compromettere entrambe le comunicazioni, allora queste passeranno obbligatoriamente attraverso questo componente. Al suo interno infatti troviamo una componente server *SE* (di tipo `ServerEvilComp`) che comunica con tutti i client compromessi. I messaggi MMS, a seconda del server su cui sono diretti, vengono inviati alla componente client corretta *CE0* o *CE1*, la quale essendo collegata con il rispettivo server DER gli potrà spedire i pacchetti. Anche in direzione opposta il server DER invierà le risposte alla corretta componente client (`CE0` o `CE1`) dell'attaccante la quale le inoltrerà alla componente server (`SE`) che si occuperà di smistarle verso l'applicazione client appropriata. Ora a causa delle limitazioni sul numero di connessioni attive contemporaneamente imposto dalle *TCPApplications* e su cui le componenti client e server dell'attaccante si fondano, lo scambio di messaggi internamente all'entità malevola non poteva essere realizzato per mezzo di connessioni TCP classiche. Per aggirare questo problema abbiamo scelto di usare il meccanismo dei **segnali**, per mezzo dei quali è possibile emettere un qualunque oggetto discendente della classe `cObject`, tra cui anche messaggi di tipo `MmsMessage`. Per l'invio dalla componente server a ciascuna delle componenti client dell'attaccante abbiamo definito una

serie di segnali `pcktFromClientSignal-*` dove grazie al supporto della *wildcard* possiamo specificare un segnale con un indice diverso per ciascuna componente client ricevente. All'opposto abbiamo creato un solo segnale `pcktFromServerSignal` per l'invio dei messaggi dalle componenti client alla componente server.
Un ulteriore aspetto da approfondire prima di procedere con l'analisi delle diverse parti di codice è quello del **comportamento dell'attaccante**. Come vedremo meglio in seguito ogni componente client possiede delle probabilità a priori di blocco o compromissione attraverso le quali è possibile stabilire se bloccare o compromettere un messaggio MMS in base alla sua tipologia. Visto che ci interessa modellare un comportamento dinamico dell'attaccante che varia in base al flusso di messaggi che riesce a monitorare, abbiamo bisogno di un formalismo più articolato. La scelta è stata quella di rappresentare questo dinamismo attraverso l'**automa a stati finiti** visibile in Figura 5.2.

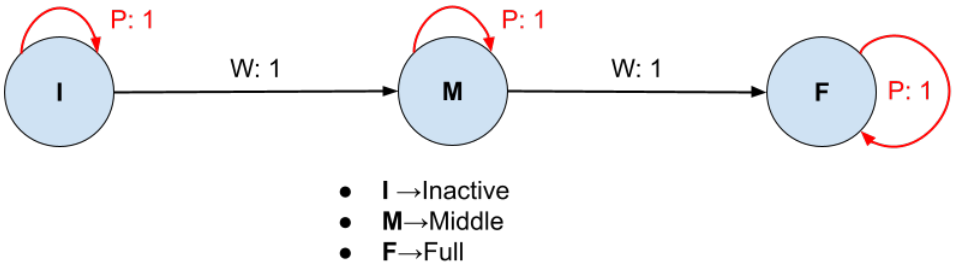


Figura 5.2: Automa a stati finiti dell'attaccante

Ogni stato modellato nella macchina possiede dei **valori d'inibizione** che definiscono di quanto devono essere ridotte le probabilità a priori di blocco o compromissione dei messaggi ($prob\_reale = prob\_priori * inib\_val$). Ad esempio nello stato *Inactive* in cui l'attaccante esegue solo attività di monitoraggio della connessione, i valori d'inibizione sono posti a 0. Nello stato intermedio *Middle* in cui l'attaccante può eseguire azioni di blocco o compromissione con probabilità dimezzata i valori d'inibizione saranno tutti a 0.5. Infine per lo stato *Full* dove l'entità malevola sferra l'attacco a intensità massima (generando anche messaggi falsi), i valori d'inibizione sono impostati a 1. A ciascun arco di transizione da uno stato al successivo (quelli rappresentati in nero) è associato un peso usato nel determinare la probabilità di cambio stato quando necessario. Il concetto di base è che il passaggio da uno stato al successivo viene determinato in base al numero di messaggi MMS transitati. All'interno dell'equazione 5.1 troviamo una **funzione sigmoide** parametrizzata in q e k.

$$s(x) = \frac{1}{1 + e^{-\frac{1}{q}x+k}} \tag{5.1}$$

Sostituendo ad x il numero di messaggi visti in un certo istante di tempo ci restituisce un valore in $[0, 1]$ che rappresenta la quantità di probabilità da distribuire sugli archi uscenti dallo stato corrente. Ad esempio inizialmente ci troviamo nel nodo *Inactive* e senza aver osservato alcun messaggio la probabilità di restare nello stato attuale è 1 (arco in rosso nell'immagine), mentre quella di passare allo stato *Middle* è 0 siccome la somma delle probabilità calcolate per tutti gli archi originati da un certo nodo deve sempre essere uguale a 1. Procedendo con la ricezione di messaggi quando

bisogna decidere se passare allo stato successivo andiamo a sottrarre alla probabilità dell'arco in rosso il valore restituito dalla sigmoide $(1 - s(x))$ e distribuiamo questa quantità sugli archi uscenti in maniera proporzionale al peso associato a ciascuno di essi (come mostrato nell'equazione 5.2).

$$P_i = \frac{s(x) * W_i}{W_{Tot}} \tag{5.2}$$

Analizzeremo meglio in seguito quando viene fatta la scelta di cambio stato. Visto lo schema generale di come le diverse applicazioni interagiscono all'interno dell'attaccante, possiamo ora addentrarci nei dettagli implementativi.

La componente server: *ServerEvilComp*

Per l'analisi della componente server iniziamo dall'osservare il file **`ServerEvilComp.ned`** visibile nel codice 5.9. Troviamo immediatamente la dichiarazione di estensione del modulo semplice `TcpGenericServerApp` da cui questo eredita alcuni parametri d'utilità necessari al funzionamento di base del protocollo e l'implementazione dell'interfaccia `IApp` (discorso già affrontato per `MmsClient` e `MmsServer`). Nella sezione `parameters` sono presenti alcune dichiarazioni di variabili come `serviceTime` (tempo di servizio impiegato nella gestione dei singoli messaggi e impostato di default secondo una distribuzione uniforme tra $10ms$ e $18ms$) e `forwardDelay` (ritardo inrodotto nell'azione d'inoltro dei messaggi alle componenti client, distribuito uniformemente tra $1us$ e $3us$). Nell'ultima riga invece troviamo la dichiarazione del segnale `pcktFromClientSignal-*` che serve a trasmettere i messaggi MMS dalla componente server alla componente client corretta.

```
1 simple ServerEvilComp extends TcpGenericServerApp {
2     parameters:
3         @class("inet::ServerEvilComp");
4 
5         volatile int serviceTime @unit(ms) = default(
              intuniform(10ms, 18ms));
6         volatile int forwardDelay @unit(us) = default(
              intuniform(1us, 3us));
7 
8         @signal[pcktFromClientSignal-*](type=Packet);
9 }
```

Codice 5.9: File `ServerEvilComp.ned`

Guardando invece al file d'intestazione **`ServerEvilComp.h`** (visibile nel codice 5.10) è possibile notare anche qui la dichiarazione di estensione della classe `TcpGenericServerApp` per l'ereditarietà di alcuni metodi necessari al funzionamento di base del protocollo. In seguito sono dichiarate alcune variabili pubbliche tra cui `messageCopier` (usato per la copia di messaggi MMS prima dell'invio su canali di comunicazione), `evilServerStatus` (variabile booleana per tenere traccia dello stato di attività del gestore dei messaggi da spedire ai client esterni), `serverQueue` (coda dei messaggi MMS da inviare ai client esterni) e `departureEvent` (evento

schedulato per il timer di gestione dei messaggi in uscita verso i client). Tra le variabili `protected` troviamo invece `forwardStatus` (variabile booleana per tenere traccia dello stato di attività del gestore degli inoltri alle componenti client interne), `forwardQueue` (coda dei messaggi MMS da inoltrare alle componenti client interne), `forwardEvent` (evento schedulato per il timer di gestione dei messaggi da inoltrare verso le componenti client), `clientCompListener` (listener con il compito di rimanere in ascolto dei messaggi provenienti dalle componenti client interne) e `pcktFromClientSignal` (vettore di `simsignal_t` che memorizza i segnali su cui emettere i messaggi diretti alle componenti client in base al loro indice). Nell'ultima parte del file troviamo i prototipi dei metodi analoghi a quelli presenti nell'`MMSServer` ma calati nel contesto della componente server dell'attaccante. Da un punto di vista funzionale non vi sono grosse variazioni ma alcune differenze si possono notare nel corpo di questi metodi.

```
1  class INET_API ServerEvilComp : public TcpGenericServerApp {
2    public:
3      MmsMessageCopier* messageCopier;
4      //Server and queue management
5      bool evilServerStatus;
6      cQueue serverQueue;
7      cMessage* departureEvent;
8
9      virtual ~ServerEvilComp();
10
11   protected:
12     //Server and queue management
13     bool forwardStatus;
14     cQueue* forwardQueue;
15     cMessage* forwardEvent;
16     // Internal signals and channel management
17     FromServerListener* clientCompListener;
18     simsignal_t* pcktFromClientSignal;
19
20   protected:
21     virtual void initialize(int stage) override;
22     virtual void handleMessage(cMessage *msg) override;
23     virtual void finish() override;
24     virtual void handleDeparture();
25     virtual void sendPacketDeparture(const MmsMessage*
         appmsg);
26     void handleForward();
27 };
```

Codice 5.10: File `ServerEvilComp.h`

Nel file **`ServerEvilComp.cc`** (visibile nel codice 5.11) troviamo per prima cosa il metodo **`initialize`** (da riga 3 a riga 38) che compie l'inizializzazione multi stadio del modulo. Eseguendo l'avvio del livello applicativo vengono per prima cosa assegnate le variabili necessarie al funzionamento del socket TCP (`localAddress`,

`localPort`) e quest'ultimo viene impostato su questi parametri e impostato in modalità "ascolto" per la ricezione di nuove connessioni. Dopo aver inizializzato altre variabili relative a messaggi di timer, stato di operatività e code di messaggi, avviene la generazione dei segnali fondamentali per la comunicazione con le componenti client. Dopo aver ottenuto il numero di applicazioni presenti dal parametro `numApps` del nodo genitore, viene inizializzato il vettore `pcktFromClientSignal` con una app in meno del valore ottenuto (vedere Figura 5.1 siccome una delle app nel nodo è configurata con la componente server corrente, mentre solo le altre saranno componenti client) e in un ciclo `for` vengono generati i nomi dei vari segnali ciascun con l'indice corretto (`pcktFromClientSignal-%d`), per poi andarli a registrare e assegnare alla corretta posizione del vettore con la funzione `registerSignal`. Dopodiché è necessario costruire il listener `FromServerListener` per poterlo iscrivere alla ricezione di messaggi da parte delle componenti client. Tale operazione è svolta dal metodo `subscribe`, che prendendo in input il listener e il nome del segnale da associargli, fa si che ogni scatto del segnale `pcktFromServerSignal` sia rilevato dalla nostra logica. Degno di nota è il fatto che il metodo subscribe sia chiamato sul nodo genitore che contiene la componente server corrente (quello che identificheremmo come *HE* in Figura 5.1) e non ad esempio alla rete principale proprio per far si che nel caso volessimo inserire diversi attaccanti nella simulazione, ciascuno vedrebbe esclusivamente i segnali relativi ai suoi componenti interni e non quelli emessi dagli altri (propagazione dei segnali nella gerarchica di cui abbiamo discusso in sezione 4.1.1). Nell'ultima parte dell'inizializzazione vengono impostate tutte quelle variabili che servono alla corretta esecuzione del ciclo di vita del modulo (avvio, operatività e spegnimento), nell'ottica di poterle sfruttare per la gestione dinamica del suo funzionamento (accenderlo e spegnerlo in maniera predeterminata). Il metodo **`handleMessage`** (da riga 40 a riga 74) viene invece chiamato da *OMNeT++* quando è necessario gestire un messaggio o un evento diretto al modulo corrente. Se il messaggio estratto dalla FEL è il `departureEvent` viene chiamato il metodo d'invio di un nuovo messaggio verso i client esterni (`handleDeparture`), mentre se è un `forwardEvent` allora viene eseguita la procedura d'inoltro di un nuovo messaggio alla componente client corretta (`handleForward`). Se nessuna delle condizioni precedenti è verificata allora ci troviamo nella situazione di dover gestire un pacchetto giunto dal socket TCP, le cui tipologie (`msg->getKind()`) e i conseguenti comportamenti possono differire:

- `TCP_I_PEER_CLOSED`: Un client connesso ha scelto di chiudere la connessione TCP, quindi il comportamento è analogo a quello tenuto nell'`MmsServer`.
- `TCP_I_DATA` o `TCP_I_URGENT_DATA`: È stato ricevuto un messaggio da parte di un client esterno, ma invece di rispondergli come fatto nel `MmsServer` ci avviamo alla gestione di un evento di forward verso una componente client. Se nessun altro messaggio è in fase d'inoltro (`forwardStatus == false`) scheduliamo il `forwardEvent` per essere gestito sulla base del `forwardDelay` configurato nel file NED e inseriamo il messaggio arrivato nella `forwardQueue` (operazione che dovremmo fare anche se il servitore delle forward fosse già occupato).
- `TCP_I_AVAILABLE`: Il socket notifica la sua operatività così che il livello applicativo memorizzi di poterne fare uso.

Se il pacchetto ricevuto non ricade in nessuna delle casistiche precedenti allora viene scartato con un messaggio di errore.
Come abbiamo già accennato il metodo **handleDeparture** (visibile da riga 76 a riga 104) si occupa di gestire le risposte da inviare ai client compromessi. Per fare ciò per prima cosa estrae il prossimo messaggio dalla `serverQueue` (`Packet *packet`), memorizza l'id della connessione ed emette un segnale di messaggio ricevuto. A questo punto nel ciclo `while` di lettura del pacchetto (avviene la conversione da `Packet` a `MmsMessage`) indipendentemente che sia di tipo `MEASURE` o di tipo `GENRESP` il messaggio viene passato al metodo `sendPacketDeparture` per essere spedito con i parametri adeguati. Al termine di queste operazioni se la `serverQueue` non è vuota viene programmato l'evento d'invio del prossimo messaggio (inserendo nuovamente il `departureEvent` nella FEL in base al `serviceTime` configurato), altrimenti il `forwardStatus` è impostato a `false`.
Abbiamo appena chiarito dove il metodo **sendPacketDeparture** venga richiamato all'interno del codice ma non ne abbiamo ancora analizzato il contenuto. Guardando da riga 106 a riga 112 è facile notare come in seguito alla generazione di un nuovo pacchetto TCP (`Packet* outPacket`) venga impostato il `socketId` prendendolo da `appmsg->getEvilServerConnId()`. Questo aspetto è particolarmente importante siccome il parametro `evilServerConnId` contiene l'identificativo della connessione tra la componente server dell'attaccante e il client compromesso. Tale valore viene memorizzato all'interno di questa variabile per evitare di perderlo durante lo split della connessione TCP. Quindi quando il messaggio torna indietro dalle componenti client dobbiamo impostare l'id della connessione al valore originale per spedire il messaggio al client compromesso corretto. A questo punto viene generata una copia del messaggio attraverso un oggetto di tipo `MessageCopier` in modo da trasferire la proprietà del messaggio al modulo corrente e poterlo spedire nuovamente sul canale di uscita (se un modulo non è proprietario del messaggio questa operazione viene impedita da *OMNeT++*). In ultima battuta l'invio del messaggio viene programmato introducendo un certo ritardo (`replyDelay`) che però nelle nostre simulazioni è stato sempre impostato a 0. L'ultima funzione del file `ServerEvilComp.cc` è la **handleForward** (da riga 116 a riga 135) che si occupa di gestire gli inoltri dei messaggi MMS verso le componenti client. Dopo aver estratto il prossimo messaggio da inoltrare dalla `forwardQueue` avviene la fase di decodifica e conversione all'interno del ciclo `while` e viene generata una copia del messaggio MMS attraverso il `messageCopier`, la quale viene poi incapsulata all'interno di un oggetto di tipo `Packet`. La componente client a cui spedire il messaggio viene determinata in base al parametro `serverIndex` del pacchetto ricevuto. Questo contiene l'indice della `ClientEvilComp` a cui deve essere inoltrato il messaggio (tale valore corrisponde per costruzione all'indice del server con cui il client compromesso era intenzionato a comunicare) e grazie a esso possiamo selezionare il segnale corretto dall'apposito vettore (`pcktFromClientSignal[appmsg->getServerIndex()]`). In seguito all'azione d'inoltro (tramite il metodo `emit`), se la `forwardQueue` non è vuota viene schedulato il prossimo evento di forward (in base al valore del parametro `forwardDelay`), altrimenti la variabile `forwardStatus` è impostata a `false`.

```
1  Define_Module(ServerEvilComp);
2
3  void ServerEvilComp::initialize(int stage) {
4      cSimpleModule::initialize(stage);
5
6      if (stage == INITSTAGE_LOCAL) {
7          ...
8      }
9      else if (stage == INITSTAGE_APPLICATION_LAYER) {
10         const char *localAddress = par("localAddress");
11         int localPort = par("localPort");
12         socket.setOutputGate(gate("socketOut"));
13         socket.bind(localAddress[0] ? L3AddressResolver().
               resolve(localAddress) : L3Address(), localPort);
14         socket.listen();
15         departureEvent = new cMessage("Server Departure");
16         forwardEvent = new cMessage("Message Forward");
17         evilServerStatus = false;
18         forwardStatus = false;
19         forwardQueue = new cQueue();
20         messageCopier = new MmsMessageCopier();
21
22         int numApps = getContainingNode(this)->par("numApps"
               ).intValue();
23         pcktFromClientSignal = new simsignal_t[numApps-1];
24         for(int i = 0; i < numApps-1; i++) {
25           char strSig[30];
26           sprintf(strSig, "pcktFromClientSignal-%d", i);
27           pcktFromClientSignal[i] = registerSignal(strSig);
28         }
29         // Initialize the listener for the incoming server
               messages
30         clientCompListener = new FromServerListener(this);
31         getContainingNode(this)->subscribe("
               pcktFromServerSignal", clientCompListener);
32         cModule *node = findContainingNode(this);
33         NodeStatus *nodeStatus = node ?
               check_and_cast_nullable<NodeStatus *>(node->
               getSubmodule("status")) : nullptr;
34         bool isOperational = (!nodeStatus) || nodeStatus->
               getState() == NodeStatus::UP;
35         if (!isOperational)
36             throw cRuntimeError("This module doesn't support
                   starting in node DOWN state");
37     }
38 }
39
```

```
40 void ServerEvilComp::handleMessage(cMessage *msg)
41 {
42     if (msg == departureEvent) handleDeparture();
43     else if(msg == forwardEvent) handleForward();
44     else if (msg->isSelfMessage()) {
45         sendBack(msg);
46     }
47     else if (msg->getKind() == TCP_I_PEER_CLOSED) {
48
49         // we'll close too, but only after there's surely no
               message
50         // pending to be sent back in this connection
51         int connId = check_and_cast<Indication *>(msg)->
              getTag<SocketInd>()->getSocketId();
52         delete msg;
53         auto request = new Request("close", TCP_C_CLOSE);
54         request->addTag<SocketReq>()->setSocketId(connId);
55         sendOrSchedule(request, SimTime(par("replyDelay").
              doubleValue(), SIMTIME_MS));
56     }
57     else if (msg->getKind() == TCP_I_DATA || msg->getKind()
          == TCP_I_URGENT_DATA) {
58       // Instead of replying directly to the client we must
            emit a signal to the internal clients of the
            evilClient
59       // to forward the message to the real server
60       if(!forwardStatus) {
61         forwardStatus = true;
62         forwardQueue->insert(msg);
63         scheduleAt(simTime() + SimTime(par("forwardDelay").
              intValue(), SIMTIME_US), forwardEvent);
64       } else forwardQueue->insert(msg);
65     }
66     else if (msg->getKind() == TCP_I_AVAILABLE)
67         socket.processMessage(msg);
68     else {
69         // some indication -- ignore
70         EV_WARN << "drop msg: " << msg->getName() << ", kind
              :" << msg->getKind() << "("
71                 << cEnum::get("inet::TcpStatusInd")->
                      getStringFor(msg->getKind()) << ")\n";
72         delete msg;
73     }
74 }
75
76 void ServerEvilComp::handleDeparture() {
77     Packet *packet = check_and_cast<Packet *>(serverQueue.
          pop());
```

```
78      int connId = packet->getTag<SocketInd>()->getSocketId();
79      ChunkQueue &queue = socketQueue[connId];
80      auto chunk = packet->peekDataAt(B(0), packet->
           getTotalLength());
81      queue.push(chunk);
82      emit(packetReceivedSignal, packet);
83      ...
84      while (queue.has<MmsMessage>(b(-1))) {
85        const auto& appmsg = queue.pop<MmsMessage>(b(-1));
86          msgsRcvd++;
87          bytesRcvd += B(appmsg->getChunkLength()).get();
88          // I set the chunk length as response length because
                we must forward the data
89          B requestedBytes = appmsg->getChunkLength();
90          if(appmsg->getMessageKind() == MMSKind::MEASURE)
               sendPacketDeparture(appmsg.get());
91          else if (appmsg->getMessageKind() == MMSKind::
               GENRESP) { //Generic Response From Server
92            if (requestedBytes > B(0)) sendPacketDeparture(
                  appmsg.get());
93          }
94          else { /* Bad Request, not present in MITM */}
95      }
96      delete packet;
97
98      ...
99
100     if(serverQueue.getLength() > 0) {
101         scheduleAt(simTime() + SimTime(par("serviceTime").
               intValue(), SIMTIME_MS),
102                 departureEvent);
103     } else evilServerStatus = false;
104 }
105
106 void ServerEvilComp::sendPacketDeparture(const MmsMessage*
       appmsg) {
107     Packet *outPacket = new Packet("Generic Data",
           TCP_C_SEND);
108     outPacket->addTag<SocketReq>()->setSocketId(appmsg->
           getEvilServerConnId());
109     const auto& payload = messageCopier->copyMessage(appmsg,
            true);
110     outPacket->insertAtBack(payload);
111     sendOrSchedule(outPacket, SimTime(par("replyDelay").
           doubleValue(), SIMTIME_MS));
112 }
113
114 // Extracts the packet from the queue and unpack the MMS
```

```
    encapsulated packet inserting the connId
115 // before forwarding it to the ClientEvilComp
116 void ServerEvilComp::handleForward() {
117   Packet* pckt = check_and_cast<Packet*>(forwardQueue->pop()
        );
118   int connId = pckt->getTag<SocketInd>()->getSocketId();
119   auto chunk = pckt->peekDataAt(B(0), pckt->getTotalLength()
        );
120   ChunkQueue &queue = socketQueue[connId];
121   queue.push(chunk);
122   while (const auto& appmsg = queue.pop<MmsMessage>(b(-1),
        Chunk::PF_ALLOW_NULLPTR)) {
123     const auto& msg = messageCopier->copyMessage(appmsg.get
          (), connId, true);
124     Packet *packet = new Packet("data");
125     packet->insertAtBack(msg);
126     emit(pcktFromClientSignal[appmsg->getServerIndex()],
          packet);
127     bubble("Sent to internal client!");
128     EV_INFO << "Conn ID:" << msg->getEvilServerConnId() << "
          \n";
129
130     if(forwardQueue->getLength() > 0) {
131       scheduleAt(simTime() + SimTime(par("forwardDelay").
            intValue(), SIMTIME_US), forwardEvent);
132     } else forwardStatus = false;
133   }
134   delete pckt;
135 }
```

Codice 5.11: File `ServerEvilComp.cc`

Proprio grazie al **`FromServerListener`** la componente server è in grado di ricevere messaggi provenienti dalle componenti client. Come possiamo vedere nel codice 5.12 nel costruttore viene memorizzato un riferimento all'oggetto genitore (`ServerEvilComp`) che viene poi usato nel metodo `receiveSignal`. Questa funzione viene chiamata nel momento in cui scatta un segnale a cui il listener è stato sottoscritto (operazione svolta in fase d'inizializzazione della componente server). Alla ricezione di un nuovo messaggio, se la componente server non sta lavorando viene programmato immediatamente un `departureEvent` a distanza `serviceTime` dal tempo di simulazione corrente, altrimenti il pacchetto viene soltanto accodato alla `serverQueue`.

```
1 FromServerListener::FromServerListener(ServerEvilComp*
    parent) {
2   this->parent = parent;
3   FromServerListener();
4 }
5
```

```
6  ...
7
8  void FromServerListener::receiveSignal(cComponent *source,
       simsignal_t signalID, cObject* value, cObject *obj) {
9    this->parent->bubble("Message from server!");
10   Packet* pckt = check_and_cast<Packet*>(value);
11   if(!this->parent->evilServerStatus) {
12     this->parent->evilServerStatus = true;
13     this->parent->serverQueue.insert(pckt);
14     this->parent->scheduleAt(simTime() + SimTime(this->
         parent->par("serviceTime").intValue(), SIMTIME_MS),
         this->parent->departureEvent);
15   }
16   else this->parent->serverQueue.insert(pckt);
17 }
```

Codice 5.12: File `FromServerListener.cc`

Le componenti client: *ClientEvilComp*

Passando allo studio delle componenti client presenti nell'attaccante iniziamo dall'osservazione del file **`ClientEvilComp.ned`** nel codice 5.13. Dopo la specifica di estensione del modulo semplice `TcpBasicClientApp` sono dichiarati diversi parametri tra cui: `fakeGenReqThresh` (numero di messaggi ogni quando generarne uno fasullo se ci si trova nello stato di azione massima), `isLogging` (flag per indicare se la componente client deve registrare su file dati sui pacchetti che la attraversano), `startFull` (variabile booleana che indica se l'attaccante debba iniziare immediatamente l'attacco nello stato *Full*), `stateChangeDelay` (intervalo di tempo tra un controllo di cambio stato e il successivo) e `checkEveryK` (numero di pacchetti ogni quanto fare un tentativo di avanzamento di stato). Inoltre da riga 17 a riga 28 troviamo diversi parametri relativi alle probabilità di blocco e compromissione di misure periodiche, richieste o risposte di lettura di variabili ed esecuzione di comandi. Nell'ultima parte del file (da riga 31 fino al fondo) sono invece dichiarate diverse coppie segnale (`@signal`) statistica (`@statistic`) relative al blocco e alla compromissione dei diversi tipi di messaggi. In particolare il primo segnala quando si verifica l'evento d'interesse a cui è associato mentre il secondo ne traccia il conteggio durante la simulazione (sfrutta il meccanismo di associazione di una statistica a un segnale sorgente).

```
1  simple ClientEvilComp extends TcpBasicClientApp {
2      parameters:
3          @class("inet::ClientEvilComp");
4
5          volatile int fakeGenReqThresh = default(10);
6
7          // For logging purposes
8          bool isLogging = default(false);
9
```

```
10          // Makes the ClientComp start and stay in "Full"
               state
11          bool startFull = default(false);
12          // Delay to try to change state
13          volatile int stateChangeDelay @unit(s) = default(5s)
               ;
14          // Number of messages to wait between a check and
               the next
15          volatile int checkEveryK = default(5);
16
17          volatile double readResponseBlockProb = default(0.1)
               ;
18          volatile double readResponseCompromisedProb =
               default(0.6);
19          volatile double commandResponseBlockProb = default
               (0.1);
20          volatile double commandResponseCompromisedProb =
               default(0.6);
21
22          volatile double readRequestBlockProb = default(0.2);
23          volatile double readRequestCompromisedProb = default
               (0.8);
24          volatile double commandRequestBlockProb = default
               (0.2);
25          volatile double commandRequestCompromisedProb =
               default(0.8);
26
27          volatile double measureBlockProb = default(0.15);
28          volatile double measureCompromisedProb = default
               (0.4);
29
30
31          @signal[genericFakeReqResSignal](type=bool);
32          @statistic[genericFakeReqRes](source=
               genericFakeReqResSignal; record=count);
33
34          @signal[measureBlockSignal](type=bool);
35          @statistic[measureBlock](source=measureBlockSignal;
               record=count);
36          @signal[measureCompromisedSignal](type=bool);
37          @statistic[measureCompromised](source=
               measureCompromisedSignal; record=count);
38
39          @signal[readRequestBlockSignal](type=bool);
40          @statistic[readRequestBlock](source=
               readRequestBlockSignal; record=count);
41          @signal[readRequestCompromisedSignal](type=bool);
42          @statistic[readRequestCompromised](source=
```

```
            readRequestCompromisedSignal; record=count);
43
44          @signal[commandRequestBlockSignal](type=bool);
45          @statistic[commandRequestBlock](source=
               commandRequestBlockSignal; record=count);
46          @signal[commandRequestCompromisedSignal](type=bool);
47          @statistic[commandRequestCompromised](source=
               commandRequestCompromisedSignal; record=count);
48
49          @signal[readResponseBlockSignal](type=bool);
50          @statistic[readResponseBlock](source=
               readResponseBlockSignal; record=count);
51          @signal[readResponseCompromisedSignal](type=bool);
52          @statistic[readResponseCompromised](source=
               readResponseCompromisedSignal; record=count);
53
54          @signal[commandResponseBlockSignal](type=bool);
55          @statistic[commandResponseBlock](source=
               commandResponseBlockSignal; record=count);
56          @signal[commandResponseCompromisedSignal](type=bool)
               ;
57          @statistic[commandResponseCompromised](source=
               commandResponseCompromisedSignal; record=count);
58
59          @signal[pcktFromServerSignal](type=cObject*);
60  }
```

Codice 5.13: File `ClientEvilComp.ned`

Passando al file d'intestazione **`ClientEvilComp.h`** (visibile nel codice 5.14), in seguito alla dichiarazione di estensione della classe `TcpBasicClientApp` possiamo notare un `messageCopier` (oggetto di tipo `MessageCopier*` con funzionalità di copia dei messaggi MMS per il trasferimento della proprietà al modulo corrente), una `msgQueue` (coda contenente i pacchetti in attesa di essere inoltrati al server), una `previousResponseSent` (variabile booleana che memorizza lo stato del servitore) e un `sendMsgEvent` (detiene un `cMessage*` relativo al timer d'invio dei messaggi accodati). Le variabili definite da riga 9 a riga 19 contengono i segnali definiti nel file NED, mentre quelle visibili da riga 21 a riga 32 memorizzano le probabilità di blocco o compromissione dei diversi tipi di pacchetti MMS impostate in fase di configurazione. Seguono due variabili con funzionalità di logging tra cui il flag di abilitazione `isLogging` e un oggetto di tipo `EvilLogger*` che si occupa di registrare i messaggi su file. Da riga 40 a riga 45 sono presenti le dichiarazioni che servono al funzionamento della macchina a stati finiti tra cui `changeStateEvent` (messaggio associato al timer di tentativo di cambio stato), `evilFSM` (riferimento alla macchina a stati finiti dell'attaccante), `startFull` (booleano per variare lo stato iniziale della FSM) e `checkEveryK` (numero k di messaggi ogni quanto controllare se passare allo stato successivo). Riguardo alle variabili `protected` sono presenti `queue` (una `ChunkQueue` utile alla decodifica dei messaggi MMS ricevuti), `serverCompListener` (listener agganciato al segnale d'inoltro dei messaggi

da parte sel `ServerEvilComp`) e `pcktFromServerSignal` (segnale per l'invio dei messaggi alla componente server). Non ci dilungheremo nell'analisi dei prototipi dei metodi siccome sono analoghi a quelli già visti per l'`MmsClient`, ma preferiamo studiarne il comportamento nel rispettivo file sorgente.

```
1  class ClientEvilComp : public TcpBasicClientApp {
2      public:
3          MmsMessageCopier* messageCopier;
4          cQueue msgQueue;
5          bool previousResponseSent;
6          cMessage* sendMsgEvent;
7
8          // Signals management
9          simsignal_t genericFakeReqResSignal;
10         simsignal_t measureBlockSignal;
11         simsignal_t measureCompromisedSignal;
12         simsignal_t readResponseBlockSignal;
13         simsignal_t readResponseCompromisedSignal;
14         simsignal_t commandResponseBlockSignal;
15         simsignal_t commandResponseCompromisedSignal;
16         simsignal_t readRequestBlockSignal;
17         simsignal_t readRequestCompromisedSignal;
18         simsignal_t commandRequestBlockSignal;
19         simsignal_t commandRequestCompromisedSignal;
20
21         double readResponseBlockProb;
22         double readResponseCompromisedProb;
23         double commandResponseBlockProb;
24         double commandResponseCompromisedProb;
25
26         double readRequestBlockProb;
27         double readRequestCompromisedProb;
28         double commandRequestBlockProb;
29         double commandRequestCompromisedProb;
30
31         double measureBlockProb;
32         double measureCompromisedProb;
33
34         // For logging purposes
35         bool isLogging;
36         EvilLogger* logger;
37
38
39         // Evil FSM to keep track of the evilServer'state
40         cMessage* changeStateEvent;
41         EvilFSM* evilFSM;
42         bool startFull;
43
```

```
44          // Number of messages to wait between two
               consecutive state checks
45          int checkEveryK;
46
47          virtual void rescheduleAfterOrDeleteTimer(simtime_t
               d, short int msgKind) override;
48          virtual int getConnectionState();
49              virtual ~ClientEvilComp();
50
51
52      protected:
53          ChunkQueue queue;
54          FromClientListener* serverCompListener;
55
56          // Internal channel signals
57          simsignal_t pcktFromServerSignal;
58
59      protected:
60          virtual void initialize(int stage) override;
61          virtual void sendRequest() override;
62          virtual void handleTimer(cMessage* msg) override;
63          virtual void socketDataArrived(TcpSocket *socket,
               Packet *msg, bool urgent) override;
64          virtual void socketEstablished(TcpSocket *socket)
               override;
65  };
```

Codice 5.14: File `ClientEvilComp.h`

È proprio il file **`ClientEvilComp.cc`** che contiene le definizioni dei corpi dei metodi che ci interessa studiare. All'interno della funzione **`initialize`** si comincia dall'inizializzare alcune variabili relative ai tempi d'inizio e di fine della comunicazione (con controlli sulla loro validità). Successivamente viene creato un nuovo riferimento a `stateChangeDelay` che viene programmato con il metodo `scheduleAt` in base alla durata di tempo contenuta nel parametro `stateChangeDelay`. Vengono poi inizializzate `startFull` e `checkEveryK` per poi creare una nuova istanza della `EvilFSM`. Nel caso il logging sia abilitato andiamo a inizializzare l'`EvilLogger` a cui viene passato il numero della run proprio per generare i log in una cartella diversa in base numero d'esecuzione. Da riga 26 a riga 37 avviene la registrazione tramite identificativo dei segnali di blocco e compromissione dei messaggi MMS di misure e comandi, mentre da riga 39 a riga 65 ne vengono inizializzate e validate le rispettive probabilità. Nella porzione finale del metodo viene creato un nuovo `FromClientListener` che viene iscritto al segnale `packetFromClientSignal-%d` dove la parte terminale della stringa è determinata dalla posizione della componente client corrente nel vettore delle app dell'host (`this->getIndex()`). La funzione **`handleTimer`** ha il compito di gestire i *self-messages* per eseguire le operazioni appropriate di risposta. Se il timer era un `changeStateEvent` allora andiamo a tentare un cambio di stato chiamando il metodo `next` sulla `evilFsm` e riprogrammiamo l'evento di cambio

stato come fatto in fase d'inizializzazione. Se non ci troviamo in questa situazione allora andiamo a controllare il tipo del messaggio che può essere:

- `MSGKIND_CONNECT`: Il socket TCP è diventato disponibile quindi possiamo inviare una richiesta di connessione (`SYN`) al server con cui la componente client si deve connettere.
- `MSGKIND_SEND`: Invia al server il prossimo messaggio MMS tramite il metodo `sendRequest`.

Una volta richiamata la funzione **`sendRequest`** estrae il prossimo messaggio dalla `msgQueue` e dopo averne generato una copia grazie al `messageCopier` lo incapsula in un nuovo pacchetto e lo spedisce. Se vi sono altri messaggi in attesa di essere inviati viene calcolato un altro tempo di servizio (`thinkTime`), il quale viene reinserito nella FEL dal metodo `rescheduleAfterOrDeleteTimer` con tipo `MSGKIND_SEND`. Il metodo **`socketEstablished`** viene invece richiamato quando la connessione TCP con il server è stata stabilita, quindi se nella `msgQueue` sono presenti messaggi giunti dalla componente server, si inizia a inviarli proprio come fatto in `sendRequest`. L'ultimo metodo e forse il più importante della componente client è il **`socketDataArrived`**. Al suo interno vengono prese le decisioni di blocco e compromissione dei messaggi provenienti dal server e diretti al client compromesso. Dopo aver richiesto quale sia lo stato corrente alla macchina a stati finti, ne estrae i rispettivi valori d'inibizione (`Inibs* inibs`). Se il pacchetto non contiene una richiesta di chiusura della connessione si procede con la decodifica e se abbiamo ricevuto un messaggio di risposta (di tipo `GENRESP`) a una richiesta falsa (l'`evilServerConnId` è a -1) allora non dobbiamo inoltrarla alla componente server, ma basta segnalare l'evento avvenuto e registrarlo con il logger se questa funzionalità è abilitata. Esclusa questa casistica possiamo chiamare il metodo `update` sulla `evilFSM` per aggiornare il conteggio dei messaggi e verificare la possibilità di un cambio di stato anche in base al parametro `checkEveryK`. Dopo aver generato una copia del messaggio con il `messageCopier` andiamo a determinarne il tipo così da individuare il corretto ramo `if-else` per gestirlo. Ad esempio supponiamo di ricevere una misura periodica dal server, estraiamo un valore `p` con distribuzione uniforme $[0, 1]$ e verifichiamo se `p` è inferiore della probabilità a priori di blocco della misura per il corrispettivo valore d'inibizione. Se ciò è verificato allora emettiamo un segnale di blocco della misura (`measureBlockSignal`) e impostiamo l'`atkStatus` del messaggio a `BLOCK` e lo registriamo nel log. Per il controllo sulla compromissione ci comportiamo in maniera simile solo che sottraiamo a `p` il valore della probabilità di blocco inibita, emettiamo un segnale di compromissione della misura (`measureCompromisedSignal`), impostiamo l'`atkStatus` del messaggio a `COMPR` e mettiamo a 9 il campo `data`. Non ci dilunghiamo nel ripetere questi concetti per le altre tipologie di messaggi in quanto analoghi. Nella parte finale del metodo avviene l'incapsulamento e l'invio vero e proprio del pacchetto tramite emissione del segnale `pcktFromServerSignal`.

```
1  Define_Module(ClientEvilComp);
2  ...
3  void ClientEvilComp::initialize(int stage) {
4      TcpAppBase::initialize(stage);
5      if (stage == INITSTAGE_LOCAL) {
6          startTime = par("startTime");
7          stopTime = par("stopTime");
8          if (stopTime >= SIMTIME_ZERO && stopTime < startTime
               )
9              throw cRuntimeError("Invalid startTime/stopTime
                   parameters");
10         timeoutMsg = new cMessage("timer");
11         messageCopier = new MmsMessageCopier();
12         changeStateEvent = new cMessage("Change State");
13         scheduleAt(simTime() + SimTime(par("stateChangeDelay
               ").intValue(), SIMTIME_S), changeStateEvent);
14         startFull = par("startFull").boolValue();
15         checkEveryK = par("checkEveryK").intValue();
16
17         // Initialize the evil FSM
18         evilFSM = new EvilFSM(this, startFull);
19
20         cEnvir* ev = getSimulation()->getActiveEnvir();
21         isLogging = par("isLogging");
22         if(isLogging) {
23           logger = new EvilLogger(ev->getConfigEx()->
                 getActiveRunNumber(), getIndex());
24         }
25
26         genericFakeReqResSignal = registerSignal("
               genericFakeReqResSignal");
27         pcktFromServerSignal = registerSignal("
               pcktFromServerSignal");
28         readRequestBlockSignal = registerSignal("
               readRequestBlockSignal");
29         readRequestCompromisedSignal = registerSignal("
               readRequestCompromisedSignal");
30         commandRequestBlockSignal = registerSignal("
               commandRequestBlockSignal");
31         commandRequestCompromisedSignal = registerSignal("
               commandRequestCompromisedSignal");
32         measureBlockSignal = registerSignal("
               measureBlockSignal");
33         measureCompromisedSignal = registerSignal("
               measureCompromisedSignal");
34         readResponseBlockSignal = registerSignal("
               readResponseBlockSignal");
```

```
35          readResponseCompromisedSignal = registerSignal("
               readResponseCompromisedSignal");
36          commandResponseBlockSignal = registerSignal("
               commandResponseBlockSignal");
37          commandResponseCompromisedSignal = registerSignal("
               commandResponseCompromisedSignal");
38
39          readResponseBlockProb = par("readResponseBlockProb")
               .doubleValue();
40          readResponseCompromisedProb = par("
               readResponseCompromisedProb").doubleValue();
41          commandResponseBlockProb = par("
               commandResponseBlockProb").doubleValue();
42          commandResponseCompromisedProb = par("
               commandResponseCompromisedProb").doubleValue();
43
44          readRequestBlockProb = par("readRequestBlockProb").
               doubleValue();
45          readRequestCompromisedProb = par("
               readRequestCompromisedProb").doubleValue();
46          commandRequestBlockProb = par("
               commandRequestBlockProb").doubleValue();
47          commandRequestCompromisedProb = par("
               commandRequestCompromisedProb").doubleValue();
48
49          measureBlockProb = par("measureBlockProb").
               doubleValue();
50          measureCompromisedProb = par("measureCompromisedProb
               ").doubleValue();
51          if(measureBlockProb < 0 || measureCompromisedProb <
               0 || (measureBlockProb + measureCompromisedProb)
               > 1) {
52              throw std::invalid_argument("Invalid measure
                   block/compromise probability");
53          }
54          if(readRequestBlockProb < 0 ||
               readRequestCompromisedProb < 0 || (
               readRequestBlockProb + readRequestCompromisedProb
               ) > 1) {
55              throw std::invalid_argument("Invalid read
                   request block/compromise probability");
56          }
57          if(commandRequestBlockProb < 0 ||
               commandRequestCompromisedProb < 0 || (
               commandRequestBlockProb +
               commandRequestCompromisedProb) > 1) {
58              throw std::invalid_argument("Invalid command
                   request block/compromise probability");
```

```
59          }
60          if(readResponseBlockProb < 0 ||
               readResponseCompromisedProb < 0 || (
               readResponseBlockProb +
               readResponseCompromisedProb) > 1) {
61              throw std::invalid_argument("Invalid read
                   response block/compromise probability");
62          }
63          if(commandResponseBlockProb < 0 ||
               commandResponseCompromisedProb < 0 || (
               commandResponseBlockProb +
               commandResponseCompromisedProb) > 1) {
64              throw std::invalid_argument("Invalid command
                   response block/compromise probability");
65          }
66          previousResponseSent = true;
67
68          // Initialize listener and subscribe to the
               serverComp forwarding signal
69          serverCompListener = new FromClientListener(this);
70          sendMsgEvent = new cMessage("Send message event");
71          char strSig[30];
72          sprintf(strSig, "pcktFromClientSignal-%d", this->
               getIndex());
73          getContainingNode(this)->subscribe(strSig,
               serverCompListener);
74      }
75  }
76
77  void ClientEvilComp::handleTimer(cMessage *msg) {
78      if(msg == changeStateEvent) {
79        evilFSM->next();
80        scheduleAt(simTime() + SimTime(par("stateChangeDelay")
             .intValue(), SIMTIME_S), changeStateEvent);
81      } else {
82          switch (msg->getKind()) {
83              case MSGKIND_CONNECT:
84                  connect();
85                  if (earlySend)
86                      sendRequest();
87                  break;
88
89              case MSGKIND_SEND:
90                  sendRequest();
91                  break;
92
93              default:
94                  throw cRuntimeError("Invalid timer msg: kind
```

```
                    =\%d", msg->getKind());
95          }
96      }
97  }
98
99  void ClientEvilComp::sendRequest() {
100     if(!msgQueue.isEmpty()) {
101       MmsMessage* msg = check_and_cast<MmsMessage*>(msgQueue
             .pop());
102         const auto& payload = messageCopier->copyMessage(msg
              , true);
103         Packet *packet = new Packet("data");
104         packet->insertAtBack(payload);
105
106         sendPacket(packet);
107         if(!msgQueue.isEmpty()) {
108           simtime_t d = simTime() + SimTime(round(par("
                thinkTime").doubleValue()), SIMTIME_MS);
109           // We suppose the client is already connected to
                the server, so when the data arrives we send it
                 (MSG_KIND_SEND)
110           rescheduleAfterOrDeleteTimer(d, MSGKIND_SEND);
111           previousResponseSent = false;
112         } else {
113           previousResponseSent = true;
114         }
115
116         delete msg;
117     }
118 }
119
120 ...
121
122 void ClientEvilComp::socketEstablished(TcpSocket *socket) {
123     TcpAppBase::socketEstablished(socket);
124
125     // Forward the packets waiting to be forwarded to the
          server
126     if(!msgQueue.isEmpty() && previousResponseSent) {
127       simtime_t d = simTime() + SimTime(round(par("thinkTime
            ").doubleValue()), SIMTIME_MS);
128     rescheduleAfterOrDeleteTimer(d, MSGKIND_SEND);
129     previousResponseSent = false;
130     }
131 }
132
133 int ClientEvilComp::getConnectionState() {
134   return socket.getState();
```

```
135 }
136
137 void ClientEvilComp::socketDataArrived(TcpSocket *socket,
       Packet *pckt, bool urgent) {
138     EvilState* currState = dynamic_cast<EvilState*>(evilFSM
           ->getCurrentState());
139     Inibs* inibs = currState->getInibValues();
140
141     if (socket->getState() == TcpSocket::LOCALLY_CLOSED) {
142         EV_INFO << "reply to last request arrived, closing
               session\n";
143         close();
144         return;
145     }
146     auto chunk = pckt->peekDataAt(B(0), pckt->getTotalLength
           ());
147     queue.push(chunk);
148     while (queue.has<MmsMessage>(b(-1))) {
149         const auto& appmsg = queue.pop<MmsMessage>(b(-1));
150         if(appmsg->getMessageKind() == MMSKind::GENRESP &&
               appmsg->getEvilServerConnId() == -1) {
151             // Emit signal for generic fake Req Res
152             emit(genericFakeReqResSignal, true);
153             if(isLogging) logger->log(appmsg.get(),
                   currState->getStateName(), simTime());
154             TcpAppBase::socketDataArrived(socket, pckt,
                   urgent);
155             return;
156         }
157
158         evilFSM->update(appmsg.get(), checkEveryK);
159         const auto& msg = messageCopier->copyMessage(appmsg.
               get(), appmsg->getEvilServerConnId(), true);
160         Packet *packet = new Packet("data");
161
162         bubble("Sent to internal client!");
163         EV_INFO << "Conn ID:" << msg->getEvilServerConnId()
               << "\n";
164
165         MMSKind messageKind = appmsg->getMessageKind();
166         ReqResKind reqResKind = appmsg->getReqResKind();
167
168         double p = this->uniform(0.0, 1.0);
169         if (messageKind == MMSKind::MEASURE) {
170             if (p < measureBlockProb * inibs->
                   getMeasureBlockInib()) { //Block
171                 bubble("Measure blocked");
172                 emit(measureBlockSignal, true);
```

```
173                 msg->setAtkStatus(MITMKind::BLOCK);
174                 if(isLogging) logger->log(msg.get(),
                        currState->getStateName(), simTime());
175                 delete packet;
176                 TcpAppBase::socketDataArrived(socket, pckt,
                        urgent);
177                 return;
178             } else if (p - (measureBlockProb * inibs->
                    getMeasureBlockInib()) <
                    measureCompromisedProb * inibs->
                    getMeasureCompromisedInib()){ //Compromise
179                 bubble("Measure compromised");
180                 emit(measureCompromisedSignal, true);
181                 msg->setAtkStatus(MITMKind::COMPR);
182                 msg->setData(9);
183                 if(isLogging) logger->log(msg.get(),
                        currState->getStateName(), simTime());
184             } else {
185                 bubble("Measure arrived from server");
186                 if(isLogging) logger->log(msg.get(),
                        currState->getStateName(), simTime());
187             }
188         } else if (messageKind == MMSKind::GENRESP) {
189             if(reqResKind == ReqResKind::READ) {
190                 if (p < readResponseBlockProb * inibs->
                        getReadResponseBlockInib()) { // Block
191                     bubble("Read response blocked");
192                     emit(readResponseBlockSignal, true);
193                     msg->setAtkStatus(MITMKind::BLOCK);
194                     if(isLogging) logger->log(msg.get(),
                            currState->getStateName(), simTime())
                            ;
195                     delete packet;
196                     TcpAppBase::socketDataArrived(socket,
                            pckt, urgent);
197                     return;
198                 } else if (p - (readResponseBlockProb *
                        inibs->getReadResponseBlockInib()) <
                        readResponseCompromisedProb * inibs->
                        getReadResponseCompromisedInib()) { //
                        Compromise
199                     bubble("Read response compromised");
200                     emit(readResponseCompromisedSignal, true
                            );
201                     msg->setAtkStatus(MITMKind::COMPR);
202                     msg->setData(9);
203                     if(isLogging) logger->log(msg.get(),
                            currState->getStateName(), simTime())
```

```
                    ;
204                 } else {
205                     bubble("Read response arrived from
                            server");
206                     if(isLogging) logger->log(msg.get(),
                            currState->getStateName(), simTime())
                            ;
207                 }
208             } else if(reqResKind == ReqResKind::COMMAND) {
209                 if (p < commandResponseBlockProb * inibs->
                        getCommandResponseBlockInib()) { // Block
210                     bubble("Command response blocked");
211                     emit(commandResponseBlockSignal, true);
212                     msg->setAtkStatus(MITMKind::BLOCK);
213                     if(isLogging) logger->log(msg.get(),
                            currState->getStateName(), simTime())
                            ;
214                     delete packet;
215                     TcpAppBase::socketDataArrived(socket,
                            pckt, urgent);
216                     return;
217                 } else if (p - (commandResponseBlockProb *
                        inibs->getCommandResponseBlockInib()) <
                        commandResponseCompromisedProb * inibs->
                        getCommandResponseCompromisedInib()) { //
                         Compromise
218                     bubble("Command response compromised");
219                     emit(commandResponseCompromisedSignal,
                            true);
220                     msg->setAtkStatus(MITMKind::COMPR);
221                     msg->setData(9);
222                     if(isLogging) logger->log(msg.get(),
                            currState->getStateName(), simTime())
                            ;
223                 } else {
224                     bubble("Command response arrived from
                            server");
225                     if(isLogging) logger->log(msg.get(),
                            currState->getStateName(), simTime())
                            ;
226                 }
227             }
228         }
229         packet->insertAtBack(msg);
230         packet->addTag<SocketInd>()->setSocketId(appmsg->
                getEvilServerConnId());
231
232         emit(pcktFromServerSignal, packet);
```

```
233     }
234
235     // Leave this call at the end because it deletes the
            packet
236     TcpAppBase::socketDataArrived(socket, pckt, urgent);
237 }
```

Codice 5.15: File `ClientEvilComp.cc`

Nel codice 5.16 possiamo trovare la definizione dei metodi del **FromClient Listener**. Come abbiamo potuto vedere il suo compito è quello di rimanere in ascolto dei messaggi provenienti dai client e accodarli alla coda dei pacchetti della propria componente server. Dopo il suo costruttore, dove vengono inizializzati alcuni parametri usati dagli altri metodi, vediamo subito la funzione **receiveSignal**. In seguito all'ottenimento dei valori d'inibizione dello stato corrente dalla FSM, avviene la decodifica del messaggio così al raggiungimento del `fakeGenReqThresh` e nel caso ci si trovasse nello stato di azione massima (`Full`), viene generato un nuovo messaggio falso. Nella implementazione attuale viene generata una richiesta di `READ` dove parametri come l'id della connessione, la dimensione attesa della risposta e la lunghezza del chunk vengono copiati dall'ultimo messaggio ricevuto. Il tempo di creazione è impostato al tempo di simulazione corrente, l'`atkStatus` è impostato a `FAKEGEN`, mentre l'`evilServerConnId` viene configurato a -1 in modo che il `ClientEvilComp` non inoltrerà la risposta di ritorno alla componente server. Nella parte di codice che va da riga 51 a riga 94 vengono svolte le stesse operazioni che abbiamo già commentato nel file `ClientEvilComp.cc`, solo che mentre in quel caso avveniva la scelta di blocco o compromissione dei messaggi diretti ai client compromessi, in questo caso l'azione è compiuta sui pacchetti diretti verso il server DER. Viene generato un apposito segnale, si effettua il logging (se abilitato) e il messaggio viene modificato (in particolare i campi `data` e `atkStatus`) o del tutto bloccato tenendo in considerazione le rispettive probabilità inibite. Indipendentemente che il messaggio generato sia falso, compromesso o non modificato il suo inserimento nella coda della componente client è affidato al metodo **enqueueNSchedule**. Nel caso in cui la `ClientEvilComp` non fosse attiva, tale funzione va anche a schedulare il prossimo evento d'invio di messaggio (sempre che il socket TCP con il server sia già attivo).

```
1  FromClientListener::FromClientListener(ClientEvilComp*
       parent) {
2      this->parent = parent;
3      fakeGenReqThresh = parent->par("fakeGenReqThresh").
           intValue();
4      numGenReq = 0;
5      FromClientListener();
6  }
7
8  ...
9
10 void FromClientListener::receiveSignal(cComponent *source,
       simsignal_t signalID, cObject* value, cObject *obj){
```

```
11      this->parent->bubble("Packet received from fwd signal!")
           ;
12      EvilState* currState = dynamic_cast<EvilState*>(parent->
           evilFSM->getCurrentState());
13      Inibs* inibs = currState->getInibValues();
14      EV << std::to_string(inibs->getMeasureBlockInib()) + "\n
           ";
15
16      Packet* pckt = check_and_cast<Packet*>(value);
17      auto chunk = pckt->peekDataAt(B(0), pckt->getTotalLength
           ());
18      queue.push(chunk);
19      while (queue.has<MmsMessage>(b(-1))) {
20        const auto& appmsg = queue.pop<MmsMessage>(b(-1));
21      if(appmsg->getMessageKind() == MMSKind::GENREQ) {
22        numGenReq++;
23        // If at least a certain number of generic requests
             has been sent
24        // generate a fake request for the server
25        if(numGenReq >= fakeGenReqThresh && currState->
             getStateName() == Full::getInstance()->getStateName
             ()) {
26          numGenReq = 0;
27
28          // Add to the server queue
29          MmsMessage* msg = new MmsMessage();
30          // Useless, could also not be set
31          msg->setOriginId(pckt->getId());
32          msg->setMessageKind(MMSKind::GENREQ);
33          // Set to read for now
34          msg->setReqResKind(ReqResKind::READ);
35          msg->setConnId(appmsg->getConnId());
36          msg->setExpectedReplyLength(appmsg->
               getExpectedReplyLength());
37          msg->setChunkLength(appmsg->getChunkLength());
38          // Set to -1 so the ClientEvilComp will not forward
               it to the server
39          msg->setEvilServerConnId(-1);
40          msg->setServerClose(false);
41          msg->addTag<CreationTimeTag>()->setCreationTime(
               simTime());
42          msg->setData(0);
43          msg->setAtkStatus(MITMKind::FAKEGEN);
44          if(parent->isLogging) parent->logger->log(msg,
               currState->getStateName(), simTime());
45
46          enqueueNSchedule(msg);
47        }
```

```
48      }
49
50      // Add to the server queue
51      MmsMessage* msg = messageCopier->copyMessageNorm(appmsg.
           get(), true);
52      parent->evilFSM->update(msg, parent->checkEveryK);
53      // Signal if a generic request gets blocked or
           compromised
54      double p = this->parent->uniform(0.0, 1.0);
55      if(appmsg->getMessageKind() == MMSKind::GENREQ) {
56        if(appmsg->getReqResKind() == ReqResKind::READ) {
57          if (p < this->parent->readRequestBlockProb * inibs->
               getReadRequestBlockInib()) { // Block
58            this->parent->emit(this->parent->
                 readRequestBlockSignal, true);
59            msg->setAtkStatus(MITMKind::BLOCK);
60            if(parent->isLogging) parent->logger->log(msg,
                 currState->getStateName(), simTime());
61            delete pckt;
62            delete msg;
63            return;
64          } else if (p - (this->parent->readRequestBlockProb *
                 inibs->getReadRequestBlockInib()) < this->parent
                ->readRequestCompromisedProb * inibs->
                getReadRequestCompromisedInib()) { // Compromise
65            msg->setAtkStatus(MITMKind::COMPR);
66            msg->setData(9);
67            if(parent->isLogging) parent->logger->log(msg,
                 currState->getStateName(), simTime());
68            this->parent->emit(this->parent->
                 readRequestCompromisedSignal, true);
69          } else {
70            if(parent->isLogging) parent->logger->log(msg,
                 currState->getStateName(), simTime());
71          }
72        } else if(appmsg->getReqResKind() == ReqResKind::
             COMMAND) {
73          if (p < this->parent->commandRequestBlockProb *
               inibs->getCommandRequestBlockInib()) { // Block
74            this->parent->emit(this->parent->
                 commandRequestBlockSignal, true);
75            msg->setAtkStatus(MITMKind::BLOCK);
76            if(parent->isLogging) parent->logger->log(msg,
                 currState->getStateName(), simTime());
77            delete pckt;
78            delete msg;
79            return;
80          } else if (p - (this->parent->
```

```
            commandRequestBlockProb * inibs->
            getCommandRequestBlockInib()) < this->parent->
            commandRequestCompromisedProb * inibs->
            getCommandRequestCompromisedInib()) { //
            Compromise
81          this->parent->emit(this->parent->
               commandRequestCompromisedSignal, true);
82          msg->setAtkStatus(MITMKind::COMPR);
83          msg->setData(9);
84          if(parent->isLogging) parent->logger->log(msg,
               currState->getStateName(), simTime());
85        } else {
86          if(parent->isLogging) parent->logger->log(msg,
               currState->getStateName(), simTime());
87        }
88      }
89    }
90
91    enqueueNSchedule(msg);
92    }
93
94    delete pckt;
95 }
96
97 void FromClientListener::enqueueNSchedule(MmsMessage* msg) {
98    if(this->parent->previousResponseSent && this->parent->
       getConnectionState() == TcpSocket::CONNECTED) {
99        this->parent->msgQueue.insert(msg);
100       simtime_t d = simTime() + SimTime(round(this->parent
          ->par("thinkTime").doubleValue()), SIMTIME_MS);
101       // We suppose the client is already connected to the
           server, so when the data arrives we send it (
          MSG_KIND_SEND)
102       this->parent->rescheduleAfterOrDeleteTimer(d,
          MSGKIND_SEND);
103       this->parent->previousResponseSent = false;
104    } else {
105       this->parent->msgQueue.insert(msg);
106    }
107 }
```

Codice 5.16: File `FromClientListener.cc`

L'ultimo elemento che ci serve per avere una visione completa delle `ClientEvil Comp` è la **`EvilFSM`**, macchina a stati finiti che regola il comportamento dell'attaccante durante la simulazione. Non ne vedremo il codice nella sua completezza, ma ci concentreremo solo sulle parti che permettono di evidenziare le scelte implementative fatte. Guardando al codice 5.17 possiamo notare come per prima cosa venga assegnato lo stato corrente alla FSM, che solitamente inizia da `Inactive`, ma se il

parametro `startFull` è impostato a `true` allora lo stato iniziale è `Full`. Un'altra operazione importante è quella d'inserimento degli archi di loop nel grafo degli stati (svolta da **`initLoops`**), ma capiremo meglio in seguito perché questa fase si rende necessaria. Al termine della costruzione della FSM viene eseguito il metodo `enter` dello stato iniziale, per svolgere le operazioni di entrata. La funzione **`next`** della `EvilFSM` ha lo scopo di eseguire la procedura `next` sullo stato corrente, la quale ritorna il prossimo stato. In realtà a seconda del numero di messaggi ricevuti (`numMessages`) può capitare che venga ritornato lo stato in cui si è già al momento, ma se ciò non si verifica vengono svolte le operazioni di uscita dallo stato corrente (`exit`), aggiornamento ed entrata in quello successivo (`enter`). Infine i metodi **`update`** e **`getInibValues`** permettono alla `ClientEvilComp` o al `FromClientListener` rispettivamente di aggiornare il conteggio del numero di messaggi transitati e ottenere i valori d'inibizione dello stato corrente (utili nel calcolo delle probabilità di blocco e compromissione effettive).

```
1  bool EvilFSM::isGraphInit = false;
2
3  EvilFSM::EvilFSM(ClientEvilComp* owner, bool startFull):
4    FSM(startFull ? Full::getInstance() : Inactive::
       getInstance())
5  {
6    numMessages = 0;
7    // Initialize all the state graph
8    if(startFull) Inactive::getInstance();
9
10   this->owner = owner;
11
12   // Initialize all the loop-arcs found starting from the
       Inactive state using a BFS
13   // but just if no other EvilFSM has already initialized
       the graph
14   if(!EvilFSM::isGraphInit) {
15     this->initLoops();
16     EvilFSM::isGraphInit = true;
17   }
18
19   // When the FSM is created we must execute the
20   // 'enter' routine for the initial state
21   this->getCurrentState()->enter(this);
22 }
23
24 // This method adds all the arcs that generate a loop in the
      graph (self-loop or not found through a BFS starting
     from the Inactive state)
25 void EvilFSM::initLoops() {
26
27 }
28
```

```
29 void EvilFSM::next() {
30   EvilState* currentState = dynamic_cast<EvilState*>(
         getCurrentState());
31   EvilState* nextState = dynamic_cast<EvilState*>(
         currentState->next(this));
32   if(nextState->getStateName() != currentState->getStateName
         ()) {
33     numMessages = 0;
34     getCurrentState()->exit(this);
35     setCurrentState(nextState);
36     getCurrentState()->enter(this);
37   }
38 }
39
40 void EvilFSM::update(const MmsMessage* msg, int checkEveryK)
       {
41   numMessages++;
42   EV << "Num: " + std::to_string(numMessages) + "\n";
43   if(numMessages \% checkEveryK == 0) { next(); }
44 }
45
46 Inibs* EvilFSM::getInibValues() {
47   return dynamic_cast<EvilState*>(getCurrentState())->
         getInibValues();
48 }
```

Codice 5.17: File `EvilFSM.cc`

Un componente fondamentale della `EvilFSM` sono gli stati che ne definiscono il comportamento. Nel codice 5.18 è presente la definizione della classe **`EvilState`**, la quale a sua volta estende la classe `FSMState`. Da quest'ultima eredita il concetto di **vettore delle transizioni**, ossia un array di coppie `<float, FSMState*>` che indica gli stati raggiungibili a partire da quello corrente con il relativo peso usato nel calcolo della probabilità di transizione. Proprio all'interno del metodo **`calcProb`** viene svolta questa operazione e viene calcolata dinamicamente la probabilità di passaggio su ciascun arco dato il suo peso e il numero di messaggi transitati (per maggiori dettagli fare riferimento all'equazione 5.2). In questo modo viene generato un vettore di probabilità posizionalmente associate al corrispettivo arco uscente dallo stato corrente. Tale oggetto viene infatti usato nel metodo **`next`** dello stato in cui ci si trova, in modo da ritornare in output lo stato di arrivo, proporzionalmente alla probabilità dell'arco che lo collega allo stato corrente. È anche possibile che la probabilità di uscita non sia sufficientemente alta e quindi in output venga restituito lo stato attuale.

```
1  const char* EvilState::stateNames[] = {"INACTIVE", "MIDDLE",
       "READONLY", "COMMANDONLY", "FULL"};
2
3  std::vector<float> EvilState::calcProb(int numMessages, std
       ::vector<std::pair<float, FSMState*>> trns) {
4    float tot = 0;
5    for(int i = 0; i < trns.size(); i++) { tot += trns[i].
       first; }
6
7    float redProb = 1 / (1 + exp(((-1/q)*numMessages)+k));
8    std::vector<float> probs;
9    for(int i = 0; i < trns.size(); i++) {
10     probs.push_back((redProb * trns[i].first) / tot);
11   }
12   return probs;
13 }
14
15 // This method is implemented here instead of in the
       FSMState because it needs a reference to the owner of
16 // the FSM which is known at this level of the hierarchy
17 FSMState* EvilState::next(FSM* machine) {
18   EvilFSM* evilMachine = check_and_cast<EvilFSM*>(machine);
19   ClientEvilComp* owner = evilMachine->owner;
20
21   std::vector<std::pair<float, FSMState*>> trns = this->
       getTransitions();
22   std::sort(trns.begin(), trns.end());
23   // Calculate transition probabilities
24   std::vector<float> probs = calcProb(evilMachine->
       getNumMessages(), trns);
25   float base = 0;
26   float p = owner->uniform(0.0, 1.0);
27   for(int i = 0; i < trns.size(); i++) {
28     EV << "Prob: " + std::to_string(probs[i]) + "\n";
29     std::pair<float, FSMState*>& tmp = trns[i];
30     if(tmp.second) {
31       if(p >= base && p < base + probs[i]) {
32         EV << "Change prob: " + std::to_string(base + probs[
             i]) + "\n";
33         EV << "Change p: " + std::to_string(p) + "\n";
34         return tmp.second;
35       }
36       base += p;
37     } else {
38       throw std::invalid_argument("Exception: the next state
           wasn't valid");
39     }
```

```
40
41    }
42    return this;
43  }
44
45  // Default behaviour when entering the new state (schedule
        the next state change)
46  void EvilState::enter(FSM* machine) {
47    EvilFSM* evilMachine = check_and_cast<EvilFSM*>(machine);
48  }
```

Codice 5.18: File `EvilState.cc`

All'atto pratico `EvilState` viene esteso da stati concreti effettivamente usati nella `EvilFSM`. Degli estratti di queste diverse classi sono visibili nei frammenti di codice 5.19, 5.20 e 5.21 (il primo più completo, mentre gli altri contengo solo la sezione principale). Nel primo possiamo studiare il design pattern adottato, ovvero il **singleton**. Di ogni diverso tipo di stato utilizzabile nella macchina ne esiste infatti una sola istanza ottenibile grazie al metodo **`getInstance`**. Questa funzione inizializza l'oggetto con il costruttore di default, altrimenti lo ritorna soltanto. L'ordine di costruzione degli stati dipende dall'ordine delle varie chiamate del metodo `getInstance` e ciò è influenzato dalla composizione dei vettori delle transizioni di ciascuno stato. Se nel nostro caso il primo stato a essere costruito in fase d'inizializzazione è `Inactive` allora seguiranno prima `Middle` e poi `Full`, siccome ciascuno è presente nel vettore delle transizioni dello stato precedente. Questo implica che se fosse presente un ciclo nel grafo degli stati, causeremmo un *deadlock* non risolvibile in fase di costruzione. Per questo nella `EvilFSM` è stato inserito il metodo in cui se necessario è possibile inserire, in seguito alla costruzione degli stati, tutti quegli archi che introdurrebbero un ciclo nel grafo. Come possiamo vedere ogni nodo del grafo possiede un nome che lo identifica e anche dei valori d'inibizione specifici (`Inibs`). Nell'ultima parte dell'estratto di `Inactive` sono presenti delle dichiarazioni necessarie a impedire che l'oggetto di quella classe sia clonabile o assegnabile.

```
1     ...
2     static EvilState* getInstance() {
3         static Inactive singleton;
4         return &(singleton);
5     }
6
7   private:
8     Inactive():
9         FSMState( { std::make_pair(1, Middle::getInstance())
              } ),
10        EvilState(EvilStateName::INACTIVE, new Inibs
              (0,0,0,0,0,0,0,0,0,0), {  std::make_pair(1,
              Middle::getInstance()) })
11    { }
12    Inactive(const Inactive& other);
```

```
13      Inactive& operator=(const Inactive& other);
14      ...
```

Codice 5.19: Estratto del file `Inactive.h`

```
1       ...
2       Middle():
3           FSMState( { std::make_pair(1, Full::getInstance()) }
                ),
4           EvilState(EvilStateName::MIDDLE, new Inibs
                (0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5), { std
                ::make_pair(1, Full::getInstance()) })
5       { }
6       ...
```

Codice 5.20: Estratto del file `Middle.h`

```
1       ...
2       Full():
3           FSMState( {  } ),
4           EvilState(EvilStateName::FULL, new Inibs
                (1,1,1,1,1,1,1,1,1,1), {  })
5       { }
6       ...
```

Codice 5.21: Estratto del file `Full.h`

Mentre la maggior parte dei parametri che vedremo possono essere modificati direttamente dal file di configurazione, quelli relativi a vettori delle transizioni e valori d'inibizione possono essere cambiati solo attraverso il codice C++.

5.2.6 Le configurazioni realizzate: Man In The Middle

Dopo aver studiato i singoli componenti delle nostre architetture ci concentreremo ora sull'analisi delle configurazioni di rete adottate nello studio dei diversi tipi di attacchi. Per prima cosa dobbiamo partire dalla configurazione `General` nel file `omnetpp.ini`.

```
1   # General configuration extended by default by every other
        configuration
2   [General]
3   sim-time-limit = 120s
4   ...
5   #General number of apps for every type of node (leave to 1)
6   **.client[*].numApps = 1
7   ...
8   **.server*.numApps = 1
9   **.evilClient.numApps = 1
10
11  # Clients Transport Settings
```

```
12 **.client[*].hasTcp = true
13 **.client[*].hasUdp = false
14
15 ....
16
17 # Server Transport Settings
18 **.server*.hasTcp = true
19 **.server*.hasUdp = false
20 **.evilClient.hasTcp = true
21 **.evilClient.hasUdp = false
22
23 ...
24
25 check-signals = false
26 num-rngs = 40
27 seed-set = ${runnumber}
28 repeat = 50
```

Codice 5.22: Configurazione `General`

Come visibile nel codice 5.22 la configurazione `General` contiene parametri relativi a diversi moduli nella rete, che non avrebbe senso ripetere più volte. Ad esempio dopo aver impostato il tempo di simulazione massimo (riga 3) a 120 secondi viene configurato a 1 il numero di applicazioni presenti nei `client`, nei `server` e nell'`evilClient` (si suppone che il caso di default preveda un client che comunica con un server con l'intervento malevolo dell'attaccante). Da riga 11 a riga 21 viene impostato a `true` il parametro `hasTcp` e a `false` il parametro `hasUdp` per tutti gli host nella rete (`client`, `server` e attaccante). Questo perché la versione implementata di MMS si basa sul protocollo `TCP` a livello di trasporto, mentre il modulo `UDP` non è necessario. Nell'ultima parte dopo aver disabilitato i controlli di tipo sui segnali emessi (utile per evitare alcuni warning in fase di debug) vengono configurati tutti i parametri necessari per il funzionamento dei generatori di numeri casuali. Il parametro `num-rngs` specifica quanti **generatori globali** di numeri casuali (numerati da 0 a $numRngs - 1$) debbano essere usati in ciascuna simulazione. Questo implica che per ogni run vengono generati tanti semi iniziali diversi quanti sono i generatori impostati. Il parametro `seed-set` va quindi a definire quale insieme di semi iniziali verrà usato per ciascuna ripetizione della simulazione. In particolare impostando questo parametro con la wildcard `$runnumber` andremo a usare un inseme di semi iniziali diverso per ogni ripetizione della simulazione. In realtà ogni modulo possiede al suo interno un certo numero di **generatori locali** di numeri casuali (numerati da 0 a $k - 1$) ciascuno dei quali viene mappato su un generatore globale. Di default viene applicato un *mapping identità* dove l'indice associato al generatore locale viene usato per determinare il generatore globale corrispondente. Questo garantisce che ciascun generatore presente in ogni modulo o canale possieda una stream di valori adeguatamente spaziata dalle altre, ma allo stesso tempo rende la simulazione ripetibile. In questo caso il numero di run da eseguire è configurato nel parametro `repeat` a 50, quindi eseguiremo la simulazione per 50 volte.

MMS_MITM_Base

In questa configurazione (visibile nel codice 5.23) troviamo alcuni parametri adottati da tutti gli scenari d'attacco Man In The Middle ma non in quelli di Distributed Denial of Service, che quindi possono essere specificati in una configurazione estendibile a parte. Incominciando da riga 7 a riga 10 troviamo le impostazioni per i diversi server, tra cui la classe da associare a ciascuna applicazione al loro interno (`MmsServer`) che ne definisce la logica, il tempo di risposta in `replyDelay` (impostato a $0ms$ per avere solo il `serviceTime` come unica fonte di ritardo), il tempo di servizio per ciascuna richiesta ricevuta in `serviceTime` (scelto secondo una distribuzione uniforme delimitata tra $2ms$ e $4ms$) e il tempo tra l'invio di una misura e quella successiva ai client sottoscritti in `emitInterval` (configurato fisso a $4s$). Da riga 13 a riga 19 è presente la configurazione del `client` in cui dopo aver specificato il tipo di ciascuna `app` al suo interno (`MmsClient`), per ciascuna di esse vengono impostati parametri come il `thinkTime` (tempo impiegato per la generazione di una richiesta e definito secondo una distribuzione uniforme tra $45ms$ e $70ms$), l'`idleInterval` (tempo di attesa tra la fine di una connessione e l'avvio della successiva), la `requestLength` (dimensione della richiesta impostata a 30 bytes) e la `replyLength` (dimensione attesa della risposta impostata a 15 bytes). Da riga 22 a riga 24 viene abilitato il logging per tutte le applicazioni contenute in ogni host (`client`, `server` ed `evilClient`). Questo permette di generare diversi file contenenti delle tracce di pacchetti gestiti da ciascun componente, così da poterle usare per elaborazioni successive. Infine da riga 29 a riga 38 vengono impostate le probabilità a priori di blocco e compromissione per ciascuna misura e richiesta o risposta di `read` o `command`. Queste vengono usate come spiegato in sezione 5.2.5.

```
1  # MMS Man In The Middle Base configuration extended by every
       other MMS MITM configuration
2  [MMS_MITM_Base]
3  # Internet cloud delayer config
4  **.internetCloud.ipv4Delayer.config = xmldoc("
       internetCloudDelays.xml")
5
6  ## Server Settings ##
7  **.server[*].app[*].typename = "MmsServer"
8  **.server[*].app[*].replyDelay = 0ms
9  **.server[*].app[*].serviceTime = intuniform(2ms, 4ms)
10 **.server[*].app[*].emitInterval = 4s
11
12 # Client Settings
13 **.client[*].app[*].typename = "MmsClient"
14 # Time Between MMS requests, this is evaluated for each
       request
15 **.client[*].app[*].thinkTime = intuniform(45ms, 70ms)
16 # Time Between TCP reconnect after a disconnect
17 **.client[*].app[*].idleInterval = intuniform(5s, 10s)
18 **.client[*].app[*].requestLength = 30B
19 **.client[*].app[*].replyLength = 15B
20
```

```
21 # Setup logging for the firt class
22 **.client[0].app[*].isLogging = true
23 **.server[*].app[*].isLogging = true
24 **.evilClient.app[*].isLogging = true
25
26 ...
27
28 # EvilClient Block/Compromise Probability configuration
29 **.evilClient.app[*].measureBlockProb = 0.15
30 **.evilClient.app[*].measureCompromisedProb = 0.4
31 **.evilClient.app[*].readResponseBlockProb = 0.1
32 **.evilClient.app[*].readResponseCompromisedProb = 0.6
33 **.evilClient.app[*].commandResponseBlockProb = 0.1
34 **.evilClient.app[*].commandResponseCompromisedProb = 0.6
35 **.evilClient.app[*].readRequestBlockProb = 0.2
36 **.evilClient.app[*].readRequestCompromisedProb = 0.5
37 **.evilClient.app[*].commandRequestBlockProb = 0.2
38 **.evilClient.app[*].commandRequestCompromisedProb = 0.6
39
40 ...
```

Codice 5.23: Configurazione `MMS_MITM_Base`

Non abbiamo ancora commentato la riga 4 per poterne parlare più approfonditamente ora. Ciò che viene fatto è configurare correttamente il modulo `ipv4Delayer` all'interno di un eventuale `InternetCloud` delayer, in modo che segua le regole definite nel file XML indicato. Abbiamo già visto il funzionamento e la sintassi di queste regole in sezione 4.2.1, ma in questo caso il file di riferimento è quello mostrato nel codice 5.24. In questa istanza la scelta è stata quella d'introdurre su ogni pacchetto un ritardo costante di $70ms$ più uno variabile uniformemente distribuito tra $0ms$ e $30ms$. Inoltre è anche incluso un ritardo di trasmissione dato dal *datarate* del canale secondo una distribuzione uniforme che varia tra $500Mbps$ e $1Gbps$. Per concludere abbiamo scelto d'impedire la perdita dei pacchetti che passano all'interno del modulo (specificando un'espressione con probabilità nulla che però permette di essere modificata velocemente in caso di bisogno).

```
1 <internetCloud symmetric="true">
2   <parameters>
3     <traffic src="**" dest="**" delay="0.070s + uniform(0s
        ,0.03s)" datarate="uniform(500Mbps, 1Gbps)" drop="
        uniform(0,1) &lt; 0"/>
4   </parameters>
5 </internetCloud>
```

Codice 5.24: File di configurazione del delayer: `internetCloudDelays.xml`

MMS_MITM_Wired_Base

Questa configurazione si è resa necessaria per raffinare ulteriormente alcuni scenari di attacco MITM (su reti cablate ma non solo) anche non presentati in questa

tesi. Dopo aver specificato la configurazione da estendere (`MMS_MITM_Base`) vengono impostati il numero di `client` e il numero di `server` (in `numClients` e `numServers`). Dopodiché viene configurato il timeout per le richieste di `read` e di `command` in modo da segnalare quando una risposta non è stata ricevuta (vedere sezione 5.2.2 per maggiori informazioni). In fine da riga 12 a riga 23 si va a configurare l'attaccante. Siccome sono presenti due server avremo bisogno di due applicazioni `ClientEvilComp` (la `app[0]` che comunica con il `server[0]` e la `app[1]` che comunica con il `server[1]`) e una sola applicazione `ServerEvilComp` (la `app[2]`) che comunica con il `client` compromesso. A questo punto restano da configurare il tempo di servizio (`thinkTime`) e il tempo di attesa tra una connessione e la successiva (`idleInterval`) per i `ClientEvilComp`, oltre che il tempo di risposta del `ServerEvilComp` (`replyDelay`).

```
1  # MMS MITM Wired configuration with 1 client and two servers
2  [MMS_MITM_Wired_Base]
3  extends = MMS_MITM_Base
4  # Defines the number of clients and the number of servers of
       the configuration
5  **.numClients = 1
6  **.numServers = 2
7
8  # Temporary increase the timeout for testing purposes (
       internet cloud)
9  *.client[*].app[*].resTimeoutInterval = 4s
10
11 # Sets the applications 0 and 1 used to communicate with the
       servers
12 *.evilClient.app[..1].typename = "ClientEvilComp"
13 # Sets the application 2 to communicate with the clients
14 *.evilClient.app[2].typename = "ServerEvilComp"
15
16 # Sets the service time for the ClientEvilComp
17 *.evilClient.app[..1].thinkTime = intuniform(5ms, 8ms)
18 # Time Between TCP reconnect after a disconnect (unused)
19 *.evilClient.app[..1].idleInterval = intuniform(5s, 10s)
20
21 ## EvilClient Settings ##
22 # Sets the reply delay for the ServerEvilComp
23 *.evilClient.app[2].replyDelay = intuniform(0ms, 5ms)
24 ...
```

Codice 5.25: Configurazione `MMS_MITM_Wired_Base`

MMS_MITM_Wired_Cli_Atk

In questa configurazione abbiamo voluto esplorare l'attacco MITM su rete completamente cablata (Ethernet) senza però simulare i ritardi introdotti dalla comunicazione attraverso una rete esterna.

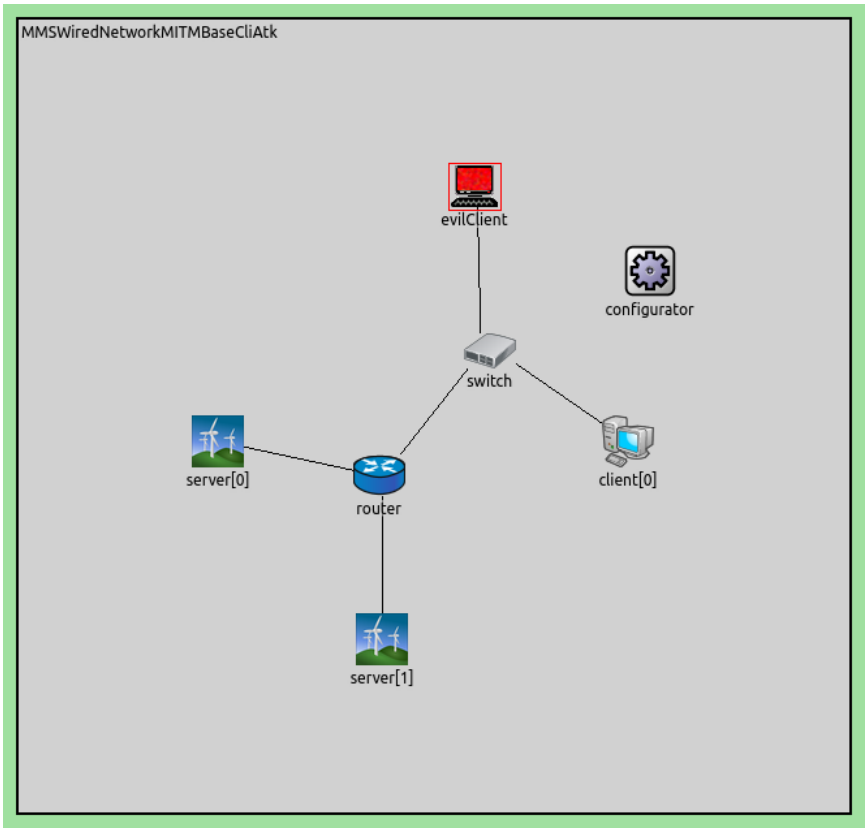


Figura 5.3: Rete `MMSWiredNetworkMITMBaseCliAtk.ned`

In Figura 5.3 possiamo notare come il `client` sia connesso sullo stesso `switch` su cui è collegato l'`evilClient` (entrambi con canale a $100Mbps$). Lo `switch` si collega a un `router` (su canale Ethernet a $1Gbps$) a cui sono collegati sia `server[0]` che `server[1]`. Il `configurator` si occupa invece di assegnare gli IP in fase d'inizializzazione e riempire le tabelle di routing. Questo scenario rappresenta il caso in cui l'attaccante abbia compromesso il client e sia riuscito a intercettare tutte le sue connessioni. Questa situazione viene modellata grazie alla configurazione mostrata nel codice 5.26, in cui dopo aver specificato i comandi di estensione della configurazione `MMS_MITM_Wired_Base` (riga 2) e di scelta dell'architettura di rete `MMSWiredNetworkMITMBaseCliAtk` (riga 3), vengono configurate le applicazioni interne al `client` per connettersi entrambe all'`evilClient` (che nei loro confronti agisce da server), il quale comprende a sua volta due client che si collegano ai corrispondenti server DER (righe 7 e 8). Abbiamo appena detto che nella configurazione, grazie al parametro `network`, viene specificato quale file NED usare per definire la struttura della rete. In realtà non ci addentreremo nella sua analisi per i diversi scenari siccome la maggior parte dei suoi componenti vengono già spiegati in fase di presentazione della figura corrispondente che è autoesplicativa. Ci sono però alcuni elementi che si ripetono in ogni rete che modella attacchi MITM e sono quelli mostrati nel codice 5.27.

```
1  [MMS_MITM_Wired_Cli_Atk]
2  extends = MMS_MITM_Wired_Base
3  network = MMSWiredNetworkMITMBaseCliAtk
4
5  # Specifies that the app[0] of the evilClient shall connect
       to the server[0] and the
6  # app[1] shall connect to the server[1].
7  *.evilClient.app[0].connectAddress = "
       MMSWiredNetworkMITMBaseCliAtk.server[0]"
8  *.evilClient.app[1].connectAddress = "
       MMSWiredNetworkMITMBaseCliAtk.server[1]"
9
10 # We simulate that both the connections with the server 0
       and server 1 are compromised
11 *.client[*].app[0].connectAddress = "
       MMSWiredNetworkMITMBaseCliAtk.evilClient"
12 *.client[*].app[1].connectAddress = "
       MMSWiredNetworkMITMBaseCliAtk.evilClient"
```

Codice 5.26: Configurazione MMS_MITM_Wired_Cli_Atk

Nella sezione parameters per prima cosa sono definiti due parametri numClients e numServers che definiscono i valori di default del numero di client e di server nella rete. Questo è fondamentale siccome questi due insiemi di componenti della rete sono rappresentati come vettori (visibili nella sezione submodules del codice), e modificando il valore delle rispettive lunghezze possiamo modellare un numero variabile di client e server nella rete. Da riga 5 a riga 18 sono invece specificate le statistiche relative alle azioni svolte dall'attaccante. Abbiamo visto in sezione 5.2.5 come ciascuna componente client dell'attaccante definisca le statistiche relative alla propria connessione compromessa, ma in questo caso ci interessa avere una visione globale, quindi andiamo a definire delle statistiche a livello di rete che agganciandosi sui segnali emessi dalle diverse componenti client dell'evilClient, effettuano un conteggio totale dei segnali a cui sono associate. In ordine abbiamo il conteggio di misure bloccate e compromesse (measureBlockGlobal e measureCompromisedGlobal), richieste e risposte di read bloccate e compromesse (readRequestBlockGlobal, readRequestCompromisedGlobal, readResponseBlockGlobal e readResponseCompromisedGlobal), richieste e risposte di command bloccate e compromesse (commandRequestBlockGlobal, commandRequestCompromisedGlobal, commandResponseBlockGlobal e commandResponseCompromisedGlobal). Infine nel vettore client il parametro numApps è impostato come il numero di server, questo perché si suppone che ogni client comunichi con tutti i server. Invece il numero di app nell'evilClient è impostato a numServers+1 per ragioni di costruzione enunciate in sezione 5.2.5.

```
1  parameters:
2      int numClients = default(1);
3      int numServers = default(2);
4
5      @statistic[measureBlockGlobal](source=measureBlockSignal
           ; record=count);
6      @statistic[measureCompromisedGlobal](source=
           measureCompromisedSignal; record=count);
7
8      @statistic[readRequestBlockGlobal](source=
           readRequestBlockSignal; record=count);
9      @statistic[readRequestCompromisedGlobal](source=
           readRequestCompromisedSignal; record=count);
10
11     @statistic[commandRequestBlockGlobal](source=
           commandRequestBlockSignal; record=count);
12     @statistic[commandRequestCompromisedGlobal](source=
           commandRequestCompromisedSignal; record=count);
13
14     @statistic[readResponseBlockGlobal](source=
           readResponseBlockSignal; record=count);
15     @statistic[readResponseCompromisedGlobal](source=
           readResponseCompromisedSignal; record=count);
16
17     @statistic[commandResponseBlockGlobal](source=
           commandResponseBlockSignal; record=count);
18     @statistic[commandResponseCompromisedGlobal](source=
           commandResponseCompromisedSignal; record=count);
19
20 submodules:
21     configurator: Ipv4NetworkConfigurator;
22     client[numClients]: StandardHost {
23         numApps = parent.numServers;
24     };
25     evilClient: StandardHost {
26         @display("i=old/comp_a");
27         numApps = parent.numServers + 1;
28     };
29     server[numServers]: StandardHost {
30         @display("i=misc/windturbine");
31     };
32     ...
```

Codice 5.27: Estratto del file `MMSWiredNetworkMITMBaseCliAtk.ned`

MMS_MITM_Wired_Cli_Atk_Delayed

Questa configurazione è pressoché analoga a quella precedente dove `client` ed `evilClient` sono connessi allo stesso `switch`, mentre `server[0]` e `server[1]` sono collegati dalla parte opposta della rete.

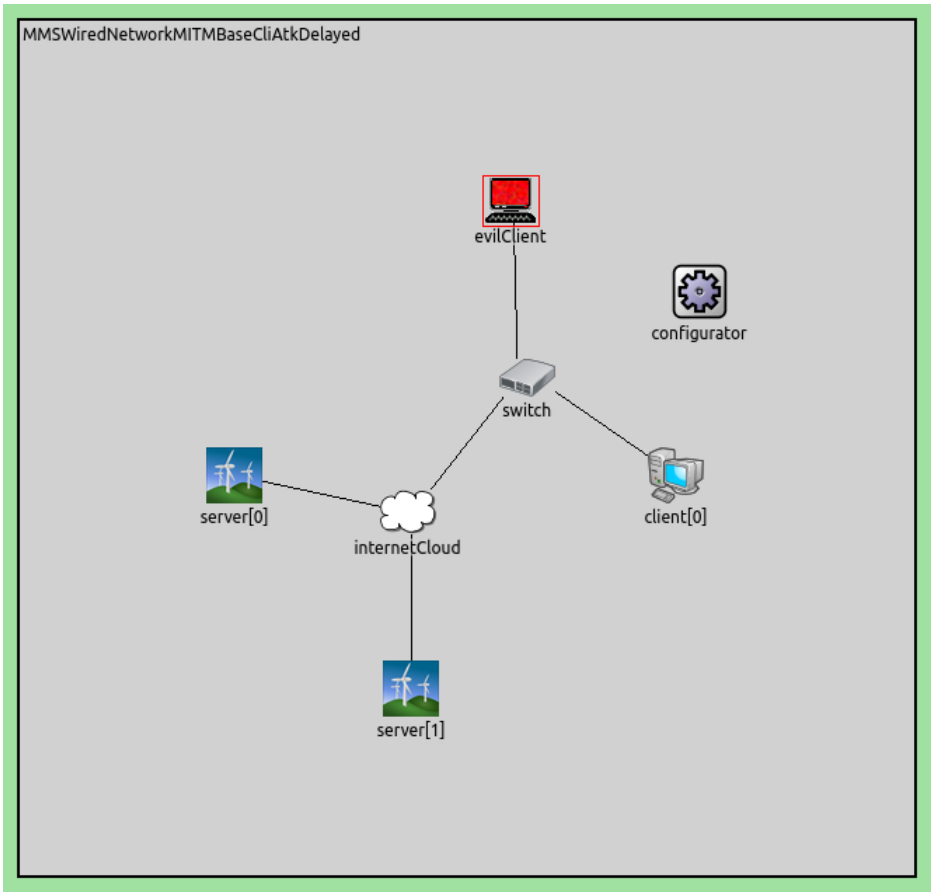


Figura 5.4: Rete `MMSWiredNetworkMITMBaseCliAtkDelayed.ned`

La differenza principale visibile in Figura 5.4 è che al posto del `router` abbiamo introdotto un `InternetCloud` delayer. Lo scopo principale di questo modulo, pur svolgendo le stesse funzioni del `router`, è quello di simulare i ritardi che avrebbero i pacchetti se passassero attraverso una rete internet classica. Al suo interno infatti possiede un modulo di `MatrixCloudDelayer` in cui sono specificate le regole riguardanti il `datarate` e il `delay` da introdurre sui singoli pacchetti (di cui abbiamo già discusso in precedenza). Al contrario in un `router` standard di *INET* non sono modellati i tempi d'instradamento, quindi seppur in quel caso si possano evidenziare meglio i ritardi introdotti dall'attaccante, non si sta operando in uno scenario realistico. Anche guardando al codice 5.28 non troviamo grosse differenze rispetto alla configurazione `MMS_MITM_Wired_Cli_Atk` se non la topologia della rete su cui si basa (`MMSWiredNetworkMITMBaseCliAtkDelayed`) e di conseguenza anche le stringhe assegnate ai parametri `connectAddress` dovranno specificare il nome della nuova rete nel *path* completo.

```
1  [MMS_MITM_Wired_Cli_Atk_Delayed]
2  extends = MMS_MITM_Wired_Base
3  network = MMSWiredNetworkMITMBaseCliAtkDelayed
4
5  # Specifies that the app[0] of the evilClient shall connect
       to the server[0] and the
6  # app[1] shall connect to the server[1].
7  *.evilClient.app[0].connectAddress = "
       MMSWiredNetworkMITMBaseCliAtkDelayed.server[0]"
8  *.evilClient.app[1].connectAddress = "
       MMSWiredNetworkMITMBaseCliAtkDelayed.server[1]"
9
10 # We simulate that both the connections with the server 0
       and server 1 are compromised
11 *.client[*].app[0].connectAddress = "
       MMSWiredNetworkMITMBaseCliAtkDelayed.evilClient"
12 *.client[*].app[1].connectAddress = "
       MMSWiredNetworkMITMBaseCliAtkDelayed.evilClient"
```

Codice 5.28: Configurazione MMS_MITM_Wired_Cli_Atk_Delayed

MMS_MITM_Wired_DER_Atk

A questo punto la scelta è stata quella di modellare lo scenario opposto, in cui l'unica connessione compromessa è quella tra `client` e `server[0]`, mantenendo comunque la rete cablata. Come visibile in Figura 5.5 l'`evilClient` ha infatti sferrato l'attacco sul `server[0]` e per questo si trova connesso sul suo stesso `switch`. Al contrario abbiamo scelto di collegare il `client` sullo stesso `router` del `server[1]` con cui invece la comunicazione procede senza anomalie. Le connessioni Ethernet hanno tutte datarate di $100Mbps$, tranne quella che collega lo `switch` al `router` che ha velocità di trasmissione di `1Gbps`.

Nella configurazione `MMS_MITM_Wired_DER_Atk` visibile nel codice 5.29 troviamo sempre il comando di estensione della configurazione `MMS_MITM_Wired_Base` e la specifica della rete da usare (`MMSWiredNetworkMITMBaseDERAtk`), ma dopo aver collegato la `app[0]` e la `app[1]` dell'attaccante rispettivamente sul `server[0]` e sul `server[1]` (righe 7 e 8) dobbiamo differenziare le connessioni fatte sul `client` (righe 12 e 13). Infatti in questo caso se la `app[0]` del `client` sarà sempre collegata all'`evilClient`, la `app[1]` dovrà connettersi direttamente al `server[1]` senza passare per l'attaccante (quella connessione non è stata compromessa).

```
1  [MMS_MITM_Wired_DER_Atk]
2  extends = MMS_MITM_Wired_Base
3  network = MMSWiredNetworkMITMBaseDERAtk
4
5  # Specifies that the app[0] of the evilClient shall connect
       to the server[1] and the
6  # app[1] shall connect to the server[1].
```

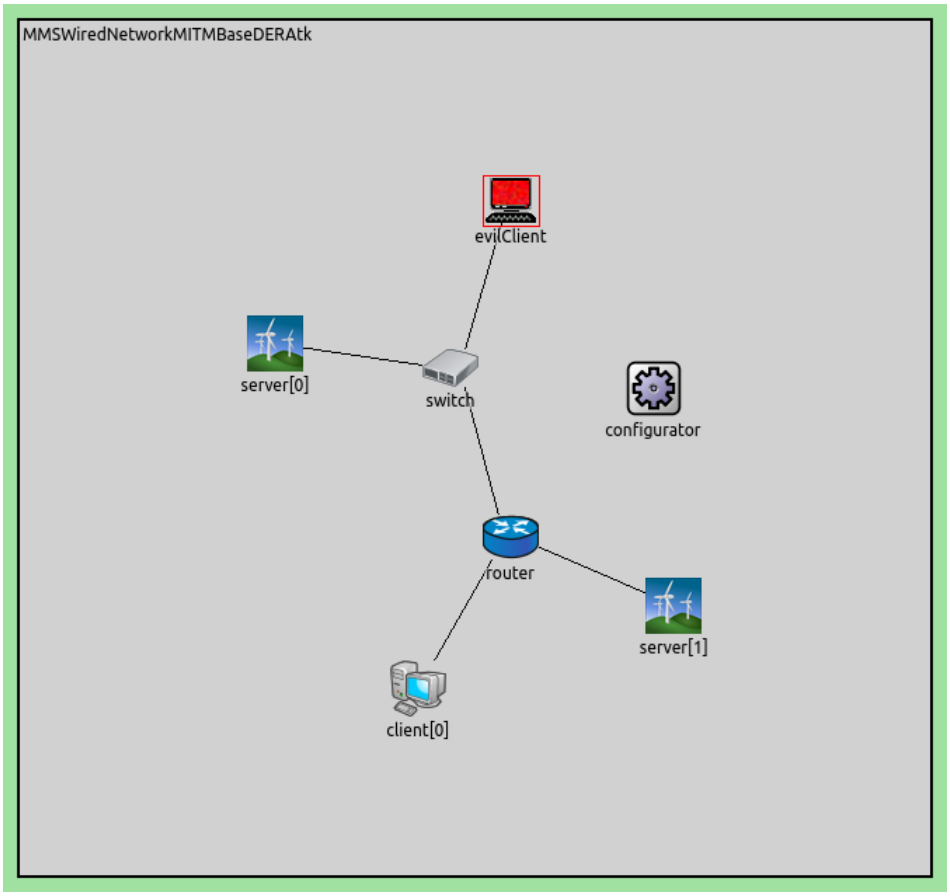


Figura 5.5: Rete `MMSWiredNetworkMITMBaseDERAtk.ned`

```
7  *.evilClient.app[0].connectAddress = "
       MMSWiredNetworkMITMBaseDERAtk.server[0]"
8  *.evilClient.app[1].connectAddress = "
       MMSWiredNetworkMITMBaseDERAtk.server[1]"
9
10 # We simulate that the connection with the server 0 is
       compromised but
11 # the connection with the server 1 is direct without the
       evilClient in between
12 *.client[*].app[0].connectAddress = "
       MMSWiredNetworkMITMBaseDERAtk.evilClient"
13 *.client[*].app[1].connectAddress = "
       MMSWiredNetworkMITMBaseDERAtk.server[1]"
```

Codice 5.29: Configurazione `MMS_MITM_Wired_DER_Atk`

MMS_MITM_Wired_DER_Atk_Delayed

Mentre nel modello precedente non ci interessava modellare i ritardi di una rete reale, in questo (visibile in Figura 5.6) abbiamo mantenuto la stessa struttura di rete ma

sostituendo il router centrale con un `InternetCloud` delayer per introdurre un ritardo ulteriore sui pacchetti. In generale sono valide tutte le affermazioni fatte per la configurazione precedente, solo con dei ritardi incrementati.

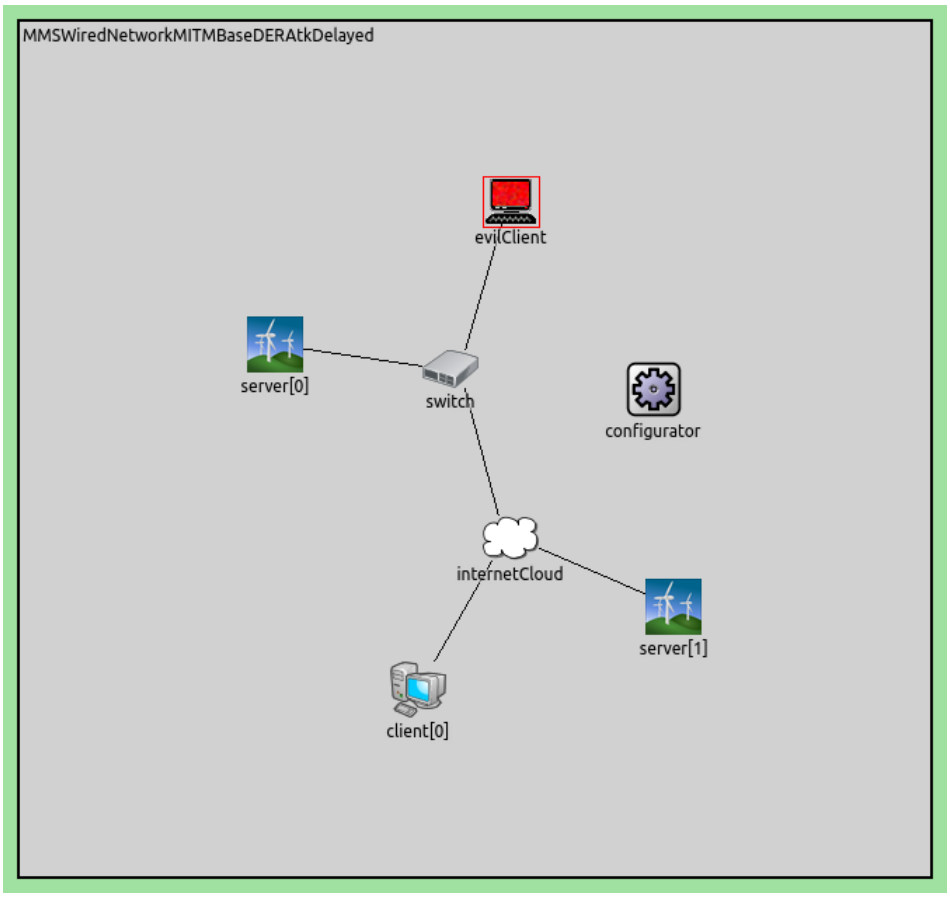


Figura 5.6: Rete `MMSWiredNetworkMITMBaseDERAtkDelayed.ned`

A livello di configurazione (visibile nel codice 5.30) viene estesa sempre la `MMS_MITM_Wired_Base`, mentre il file NED della topologia è il `MMSWiredNetwork MITMBaseDERAtkDelayed`. All'interno dei parametri `connectAddress` le impostazioni sono analoghe a quelle della configurazione `MMS_MITM_Wired_DER_Atk` ma con la prima parte del *path* che fa riferimento alla nuova rete.

```
1  [MMS_MITM_Wired_DER_Atk_Delayed]
2  extends = MMS_MITM_Wired_Base
3  network = MMSWiredNetworkMITMBaseDERAtkDelayed
4  # Specifies that the app[0] of the evilClient shall connect
       to the server[0] and the
5  # app[1] shall connect to the server[1].
6  *.evilClient.app[0].connectAddress = "
       MMSWiredNetworkMITMBaseDERAtkDelayed.server[0]"
7  *.evilClient.app[1].connectAddress = "
       MMSWiredNetworkMITMBaseDERAtkDelayed.server[1]"
8  # We simulate that the connection with the server 0 is
       compromised but
9  # the connection with the server 1 is direct without the
       evilClient in between
10 *.client[*].app[0].connectAddress = "
       MMSWiredNetworkMITMBaseDERAtkDelayed.evilClient"
11 *.client[*].app[1].connectAddress = "
       MMSWiredNetworkMITMBaseDERAtkDelayed.server[1]"
```

Codice 5.30: Configurazione `MMS_MITM_Wired_DER_Atk_Delayed`

MMS_MITM_Wireless_Cli_Atk

In questa configurazione abbiamo deciso d'introdurre una connessione Wi-Fi 802.11 all'interno della rete. Come si può vedere in Figura 5.7, `client` ed `evilClient` non sono più degli `StandardHost`, ma sono dei `WirelessHost` (analoghi ai primi ma con un'interfaccia wlan `Ieee80211Interface`) connessi in rete attraverso un `accessPoint` a cui si associano dinamicamente tramite una fase di autenticazione wireless svolta a inizio simulazione. Per inizializzarli correttamente è necessario specificare una posizione fissa nel modulo interno `mobility` (il cui tipo è `StationaryMobility`) e può essere fatto sia all'interno del file NED nella `displayString` o direttamente nella configurazione come vedremo in seguito. L'`accessPoint` è collegato a un router (tramite canale Ethernet a $1Gbps$) a cui sono collegati `server[0]` e `server[1]` (su canale Ethernet a $100Mbps$). Come per le altre configurazioni è presente un `configurator` per le operazioni d'inizializzazione degli IP e di routing, ma siccome abbiamo introdotto una comunicazione wireless è necessario anche inserire un modulo di `Ieee80211RadioMedium` che definisce la rappresentazione del mezzo di trasmissione in uso così da calcolare interferenze, rumore di fondo e raggi di trasmissione come presentato in sezione 4.2.1.

Come possiamo vedere nel codice 5.31 la configurazione estesa non cambia, mentre specifichiamo `MMSWirelessNetworkMITMBaseCliAtk` come rete che definisca la topologia della simulazione. Dopodiché a riga 5 impostiamo il modulo di mobilità dell'attaccante per prendere la posizione dalla stringa d'inizializzazione nel file NED (operazione non necessaria per `client` e `accessPoint` siccome per loro tale valore non è mai stato impostato a `false`). Infine supponendo che l'attaccante abbia compromesso tutte le connessioni del `client`, configuriamo i `connectAddress` di

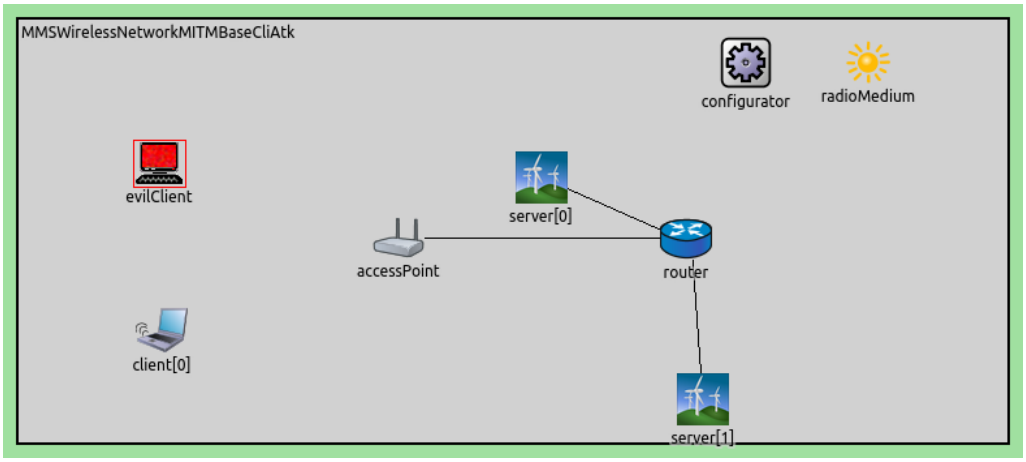


Figura 5.7: Rete `MMSWirelessNetworkMITMBaseCliAtk.ned`

`app[0]` e `app[1]` per collegarsi all'`evilClient`, il quale a sua volta connetterà le sue `app[0]` e `app[1]` a `server[0]` e `server[1]` (si usano sempre *path* completi con il nome della rete specificato).

```
1  [MMS_MITM_Wireless_Cli_Atk]
2  extends = MMS_MITM_Wired_Base
3  network = MMSWirelessNetworkMITMBaseCliAtk
4
5  *.evilClient.mobility.initFromDisplayString = true
6
7  # Specifies that the app[0] of the evilClient shall connect
       to the server[0] and the
8  # app[1] shall connect to the server[1].
9  *.evilClient.app[0].connectAddress = "
       MMSWirelessNetworkMITMBaseCliAtk.server[0]"
10 *.evilClient.app[1].connectAddress = "
       MMSWirelessNetworkMITMBaseCliAtk.server[1]"
11
12 # We simulate that both the connections with the server 0
       and server 1 are compromised
13 *.client[*].app[0].connectAddress = "
       MMSWirelessNetworkMITMBaseCliAtk.evilClient"
14 *.client[*].app[1].connectAddress = "
       MMSWirelessNetworkMITMBaseCliAtk.evilClient"
```

Codice 5.31: Configurazione `MMS_MITM_Wireless_Cli_Atk`

MMS_MITM_Wireless_Cli_Atk_Delayed

In questa variante della configurazione precedente sostituiamo il `router` centrale con un `InternetCloud` delayer per introdurre ritardi configurabili. La restante parte della topologia mostrata in Figura 5.8 è identica a quella usata nella configurazione `MMS_MITM_Wireless_Cli_Atk`.

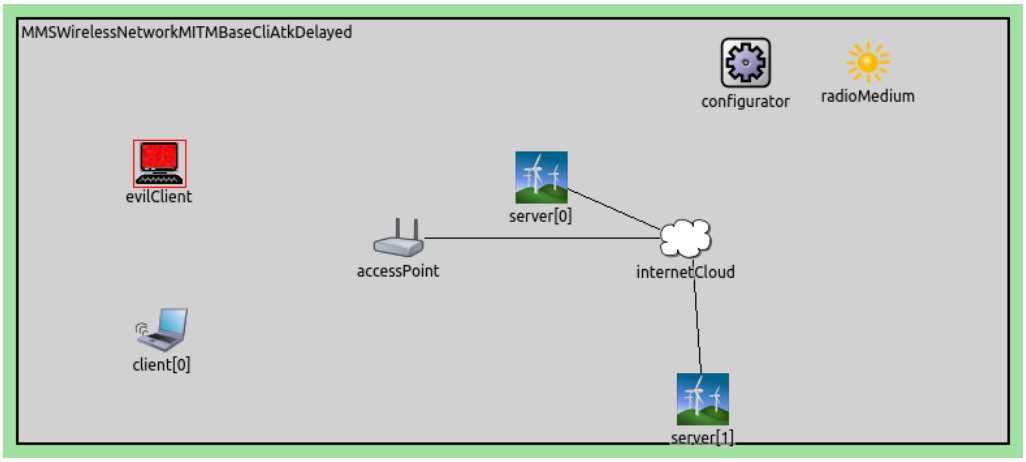


Figura 5.8: Rete `MMSWirelessNetworkMITMBaseCliAtkDelayed.ned`

Anche nella rispettiva configurazione visibile nel codice 5.32 non vi sono grossi cambiamenti se non la rete specificata nel parametro `network` (`MMSWirelessNetworkMITMBaseCliAtkDelayed`) e anche le stringhe usate nei parametri `connectAddress` che varieranno di conseguenza.

```
1  [MMS_MITM_Wireless_Cli_Atk_Delayed]
2  extends = MMS_MITM_Wired_Base
3  network = MMSWirelessNetworkMITMBaseCliAtkDelayed
4
5  *.evilClient.mobility.initFromDisplayString = true
6
7  # Specifies that the app[0] of the evilClient shall connect
       to the server[0] and the
8  # app[1] shall connect to the server[1].
9  *.evilClient.app[0].connectAddress = "
       MMSWirelessNetworkMITMBaseCliAtkDelayed.server[0]"
10 *.evilClient.app[1].connectAddress = "
       MMSWirelessNetworkMITMBaseCliAtkDelayed.server[1]"
11
12 # We simulate that both the connections with the server 0
       and server 1 are compromised
13 *.client[*].app[0].connectAddress = "
       MMSWirelessNetworkMITMBaseCliAtkDelayed.evilClient"
14 *.client[*].app[1].connectAddress = "
       MMSWirelessNetworkMITMBaseCliAtkDelayed.evilClient"
```

Codice 5.32: Configurazione `MMS_MITM_Wireless_Cli_Atk_Delayed`

MMS_MITM_Wireless_DER_Atk

Come fatto per le configurazioni cablate, anche per le configurazioni wireless abbiamo voluto modellare il caso in cui l'attaccante fosse riuscito a compromettere solo la connessione tra `client` e `server[0]`. Come possiamo vedere in Figura 5.9

il client è uno StandardHost connesso al router attraverso un canale Ethernet da 100*Mbps*. A quest'ultimo è collegato un vettore di accessPoint (tramite canali Ethernet da 1*Gbps*) di dimensione pari al numero di server presenti nella rete. In questa istanza i server sono dei WirelessHost e server[0] con l'evilClient si connettono all'accessPoint[0], mentre il server[1] si collega all'accessPoint[1]. Vedremo nel seguito come effettuare staticamente questa associazione tra host e accessPoint.

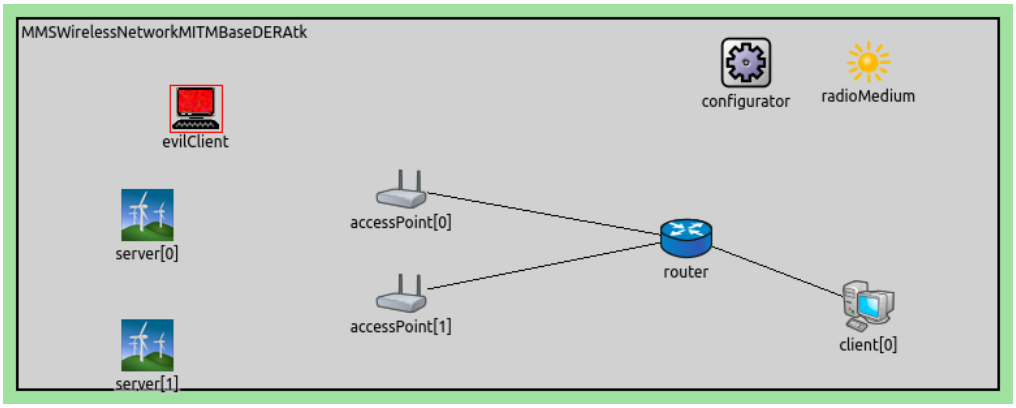


Figura 5.9: Rete MMSWirelessNetworkMITMBaseDERAtk.ned

Guardando alla configurazione nel codice 5.33 è possibile notare come la rete di riferimento sia MMSWirelessNetworkMITMBaseDERAtk e che in questo caso la app[0] del client sia connessa all'evilClient, mentre la app[1] è collegata direttamente con il server[1]. Come possiamo notare in questa configurazione e in tutte le altre su rete Wi-Fi dove l'attaccante è connesso sull'accessPoint del server, la configurazione estesa è la MMS_MITM_Wireless_DER_Atk_Base visibile nel codice 5.34.

```
1  [MMS_MITM_Wireless_DER_Atk]
2  extends = MMS_MITM_Wireless_DER_Atk_Base
3  network = MMSWirelessNetworkMITMBaseDERAtk
4
5  *.evilClient.mobility.initFromDisplayString = true
6
7  # Specifies that the app[0] of the evilClient shall connect
     to the server[0] and the
8  # app[1] shall connect to the server[1].
9  *.evilClient.app[0].connectAddress = "
     MMSWirelessNetworkMITMBaseDERAtk.server[0]"
10 *.evilClient.app[1].connectAddress = "
     MMSWirelessNetworkMITMBaseDERAtk.server[1]"
11
12 # We simulate that the connection with the server 0 is
     compromised but
13 # the connection with the server 1 is direct without the
     evilClient in between
```

```
14 *.client[*].app[0].connectAddress = "
      MMSWirelessNetworkMITMBaseDERAtk.evilClient"
15 *.client[*].app[1].connectAddress = "
      MMSWirelessNetworkMITMBaseDERAtk.server[1]"
```

Codice 5.33: Configurazione MMS_MITM_Wireless_DER_Atk

Questa infatti contiene diverse impostazioni che se non fossero specificate al suo interno dovrebbero essere ripetute sia in MMS_MITM_Wireless_DER_Atk che in MMS_MITM_Wireless_DER_Atk_Delayed. In questa configurazione per prima cosa viene specificata la posizione degli accessPoint e dei server (righe dalla 5 alla 14) impostando a false il parametro initFromDisplayString e mettendo le coordinate nei parametri initialX e initialY dei rispettivi moduli mobility di ciascun host. Nella seconda parte della configurazione (da riga 17 a riga 22) avviene la configurazione statica dell'evilClient e dei server sui corretti accessPoint. In particolare il modulo di gestione (mgmt) dell'accessPoint[0] viene configurato con SSID AP0, mentre il modulo di gestione dell'accessPoint[1] viene configurato con SSID AP1. Questo ci permette di configurare il parametro defaultSsid delle schede wireless di evilClient e server[0] per connettersi alla rete con SSID AP0, mentre il sererver[1] si assocerà con l'accessPoint il cui SSID è AP1.

```
1  [MMS_MITM_Wireless_DER_Atk_Base]
2  extends = MMS_MITM_Wired_Base
3
4  # Fixed position of the access point
5  *.accessPoint[*].mobility.initFromDisplayString = false
6  *.accessPoint[*].mobility.initialX = 294m
7  *.accessPoint[0].mobility.initialY = 130m
8  *.accessPoint[1].mobility.initialY = 210m
9
10 # Fixed positions for server
11 *.server[*].mobility.initFromDisplayString = false
12 *.server[*].mobility.initialX = 99m
13 *.server[0].mobility.initialY = 150m
14 *.server[1].mobility.initialY = 250m
15
16 # Statically binding client and server to the respective APs
17 **.accessPoint[0].wlan[*].mgmt.ssid = "AP0"
18 **.accessPoint[1].wlan[*].mgmt.ssid = "AP1"
19
20 **.evilClient.wlan[*].agent.defaultSsid = "AP0"
21 **.server[0].wlan[*].agent.defaultSsid = "AP0"
22 **.server[1].wlan[*].agent.defaultSsid = "AP1"
```

Codice 5.34: Configurazione MMS_MITM_Wireless_DER_Atk_Base

MMS_MITM_Wireless_DER_Atk_Delayed

Anche per la versione wireless dell'attacco MITM al `server[0]` abbiamo voluto realizzare una variante in cui il router venisse sostituito con un `InternetCloud` delayer (visibile in Figura 5.10).

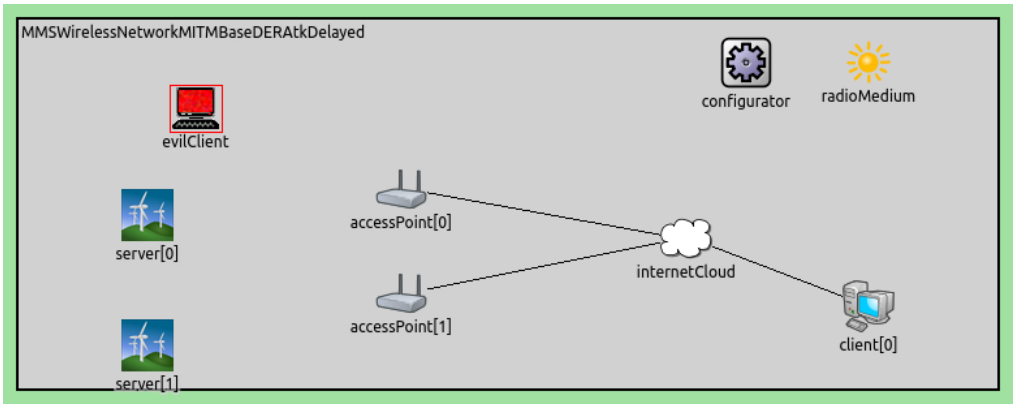


Figura 5.10: Rete `MMSWirelessNetworkMITMBaseDERAtkDelayed.ned`

Guardando alla configurazione nel codice 5.35 possiamo vedere che estende `MMS_MITM_Wireless_DER_Atk_Base` e la rete di riferimento è `MMSWireless NetworkMITMBaseDERAtkDelayed`. Per la parte restante la configurazione è identica alla controparte senza delayer, siccome modella sempre solo la connessione al `server[0]` come compromessa, mentre quella con il `server[1]` non è alterata.

```
1  [MMS_MITM_Wireless_DER_Atk_Delayed]
2  extends = MMS_MITM_Wireless_DER_Atk_Base
3  network = MMSWirelessNetworkMITMBaseDERAtkDelayed
4
5  *.evilClient.mobility.initFromDisplayString = true
6
7  # Specifies that the app[0] of the evilClient shall connect
       to the server[0] and the
8  # app[1] shall connect to the server[1].
9  *.evilClient.app[0].connectAddress = "
       MMSWirelessNetworkMITMBaseDERAtkDelayed.server[0]"
10 *.evilClient.app[1].connectAddress = "
       MMSWirelessNetworkMITMBaseDERAtkDelayed.server[1]"
11
12 # We simulate that the connection with the server 0 is
       compromised but
13 # the connection with the server 1 is direct without the
       evilClient in between
14 *.client[*].app[0].connectAddress = "
       MMSWirelessNetworkMITMBaseDERAtkDelayed.evilClient"
15 *.client[*].app[1].connectAddress = "
       MMSWirelessNetworkMITMBaseDERAtkDelayed.server[1]"
```

Codice 5.35: Configurazione `MMS_MITM_Wireless_DER_Atk_Delayed`

MMS_MITM_5G_Cli_Atk

L'ultima serie di modelli è rappresentata dalle reti in cui è stata introdotta un'infrastruttura 5G per la comunicazione dei `server` con la parte centrale della rete. Come visibile in Figura 5.11 ciascun `server` è istanziato come `NRUe`: un host che presenta un'interfaccia `cellularNic` in grado di connettersi alle reti 5G (ma anche a quelle LTE) per l'invio di pacchetti. In particolare `server[0]` e `server[1]` sono associati staticamente rispettivamente alla `gnb1` e alla `gnb2`, le quali svolgono la funzione di un'antenna che fa passare il segnale da trasmissione radio a trasmissione cablata. Entrambe le `gNodeBs` infatti sono collegate tramite canale Ethernet da $1Gbps$ a un `upf` (User Plane Function) che funge da gateway per indirizzare i pacchetti sull'antenna corretta o trasmetterli a una rete 5G diversa. Infine l'`upf` entra in comunicazione con il core della rete (in questo caso rappresentato da un `router`) attraverso il suo `filterGate`. In questo particolare modello stiamo simulando che l'attaccante sia riuscito a compromettere il `client`, quindi si collega assieme a quest'ultimo allo stesso `switch` (con canale Ethernet da $100Mbps$). Oltre al modulo globale `configurator` già presente nelle reti *INET* per svolgere la fase di assegnazione degli IP e di routing è necessario introdurne altri per il corretto funzionamento dei moduli 5G. Il `binder` detiene dei riferimenti agli altri moduli nella rete e implementa le funzioni del control plane. Il `carrierAggregation` e il `channelControl` invece servono a modellare i canali 5G e a tenere traccia delle comunicazioni in atto, così da poter tenere in considerazione delle interferenze generate da ogni trasmissione.

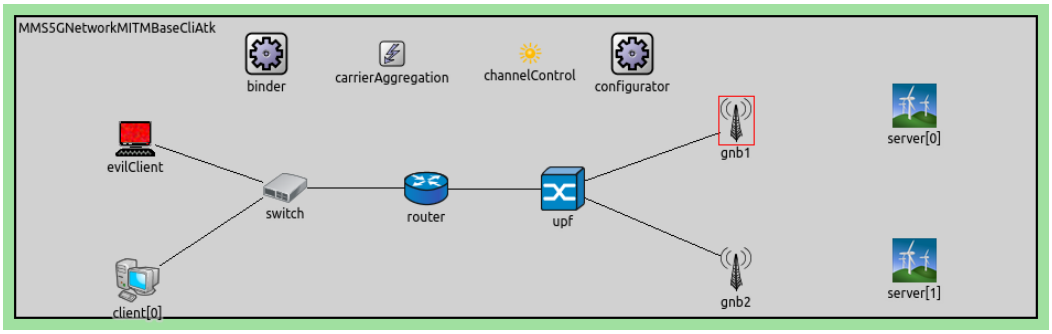


Figura 5.11: Rete `MMS5GNetworkMITMBaseCliAtk.ned`

Per ottenere la configurazione appena illustrata è necessario affidarsi al codice 5.36. Nel parametro `network` viene indicata la rete di riferimento `MMS5GNetworkMITMBaseCliAtk` e come per tutti i modelli di questa categoria, dopo aver indicato a ciascuna `app` dell'`evilClient` di connettersi al rispettivo `server`, si va a configurare ciascuna `app` del `client` per collegarsi all'`evilClient` siccome entrambe le connessioni sono state compromesse.

```
1 [MMS_MITM_5G_Cli_Atk]
2 extends = MMS_MITM_5G_Cli_Atk_Base
3 network = MMS5GNetworkMITMBaseCliAtk
4 *.evilClient.mobility.initFromDisplayString = true
5 # Specifies that the app[0] of the evilClient shall connect
     to the server[0] and the
6 # app[1] shall connect to the server[1].
```

```
7  *.evilClient.app[0].connectAddress = "
       MMS5GNetworkMITMBaseCliAtk.server[0]"
8  *.evilClient.app[1].connectAddress = "
       MMS5GNetworkMITMBaseCliAtk.server[1]"
9  # We simulate that the connection with the server 0 is
       compromised but
10 # the connection with the server 1 is direct without the
       evilClient in between
11 *.client[*].app[0].connectAddress = "
       MMS5GNetworkMITMBaseCliAtk.evilClient"
12 *.client[*].app[1].connectAddress = "
       MMS5GNetworkMITMBaseCliAtk.evilClient"
13 **.gnb*.gateway = "MMS5GNetworkMITMBaseCliAtk.upf"
```

Codice 5.36: Configurazione MMS_MITM_5G_Cli_Atk_Base

In realtà in questo tipo di reti è anche necessario configurare la porzione d'infrastruttura 5G e questo viene fatto nella configurazione MMS_MITM_5G_Cli_Atk_Base estesa da tutte le altre di questo tipo. Come visibile nel codice 5.37, dopo aver specificato la configurazione da estendere (MMS_MITM_Wired_Base) si vanno a configurare server[0] e server[1] (da riga 4 a riga 12) per connettersi rispettivamente alle gNodeBs con id 1 (gnb1) e con id 2 (gnb2). In particolare si impostano i parametri macCellId, masterId, nrMacCellId e nrMasterId per contenere l'id della cella a cui ciascun host si dovrà connettere. Tale operazione è necessaria per modellare questo scenario siccome in alternativa dovremmo abilitare l'associazione dinamica tra NRUe e gNodeB e ciò non garantirebbe l'assenza di cambi cella durante l'esecuzione. Nella parte finale della configurazione (da riga 14 a riga 17) si va invece a impostare il posizionamento statico dei server che invece di essere inizializzato dalla *display string* nel file NED, viene impostato all'interno del modulo mobility dei singoli server con i parametri initialX e initialY.

```
1  [MMS_MITM_5G_Cli_Atk_Base]
2  extends = MMS_MITM_Wired_Base
3  # Connect the NRUe's NIC to the corresponding serving gNB
4  **.server[0].macCellId = 1
5  **.server[0].masterId = 1
6  **.server[0].nrMacCellId = 1
7  **.server[0].nrMasterId = 1
8
9  **.server[1].macCellId = 2
10 **.server[1].masterId = 2
11 **.server[1].nrMacCellId = 2
12 **.server[1].nrMasterId = 2
13
14 *.server[*].mobility.initFromDisplayString = false
15 *.server[*].mobility.initialX = 815m
16 *.server[0].mobility.initialY = 80m
17 *.server[1].mobility.initialY = 220m
```

Codice 5.37: Configurazione MMS_MITM_5G_Cli_Atk_Base

MMS_MITM_5G_Cli_Atk_Delayed

Questa configurazione (visibile in Figura 5.12) presenta la struttura della `MMS_MITM_5G_Cli_Atk` ma al posto del `router` è stato introdotto un `internetCloud` delayer per simulare i ritardi della rete esterna.

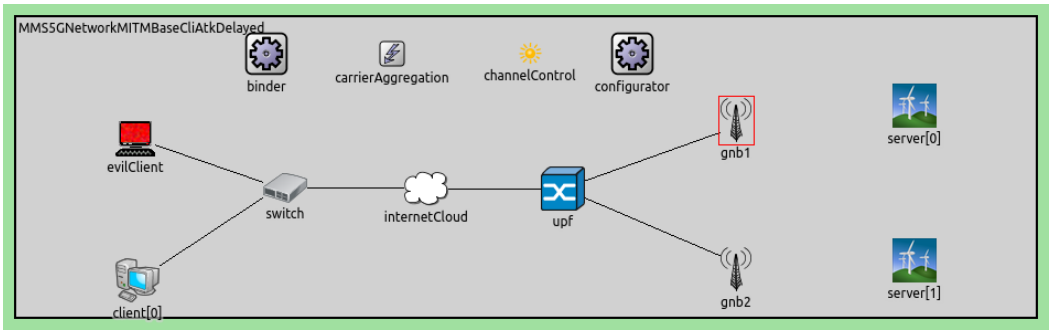


Figura 5.12: Rete `MMS5GNetworkMITMBaseCliAtkDelayed.ned`

Anche dal punto di vista della configurazione (visibile nel codice 5.38) non vi sono grosse variazioni, se non che nelle stringhe assegnate ai parametri `connectAddress` sia specificato il nome della rete usata (`MMS5GNetworkMITMBaseCliAtkDelayed`).

```
1  [MMS_MITM_5G_Cli_Atk_Delayed]
2  extends = MMS_MITM_5G_Cli_Atk_Base
3  network = MMS5GNetworkMITMBaseCliAtkDelayed
4
5  *.evilClient.mobility.initFromDisplayString = true
6
7  # Specifies that the app[0] of the evilClient shall connect
       to the server[0] and the
8  # app[1] shall connect to the server[1].
9  *.evilClient.app[0].connectAddress = "
       MMS5GNetworkMITMBaseCliAtkDelayed.server[0]"
10 *.evilClient.app[1].connectAddress = "
       MMS5GNetworkMITMBaseCliAtkDelayed.server[1]"
11
12 # We simulate that both the connections with the server 0
       and server 1 are compromised
13 *.client[*].app[0].connectAddress = "
       MMS5GNetworkMITMBaseCliAtkDelayed.evilClient"
14 *.client[*].app[1].connectAddress = "
       MMS5GNetworkMITMBaseCliAtkDelayed.evilClient"
15
16 **.gnb*.gateway = "MMS5GNetworkMITMBaseCliAtkDelayed.upf"
```

Codice 5.38: Configurazione `MMS_MITM_5G_Cli_Atk_Delayed`

MMS_MITM_5G_DER_Atk

Per concludere anche con le reti 5G abbiamo implementato lo scenario in cui l'attaccante è riuscito a compromettere solo la connessione con il `server[0]` e si trova quindi connesso tramite rete *New Radio* sulla stessa `gNodeB` di quest'ultimo (gnb1). Ciò implica che in questa nuova rete (visibile in Figura 5.13) l'`evilClient` debba essere un `NRUe` invece che uno `StandardHost` e che allo `switch` presente sia collegato esclusivamente il `client` su canale Ethernet da $100Mbps$. Come nella configurazione `MMS_MITM_5G_Cli_Atk` il `server[1]` è connesso alla `gnb2` la quale assieme a `gnb1` è collegata all'`upf` di gateway (su canale Ethernet da $1Gbps$) e il nodo centrale della rete è rappresentato da un `router` con canali uscenti da $1Gbps$.

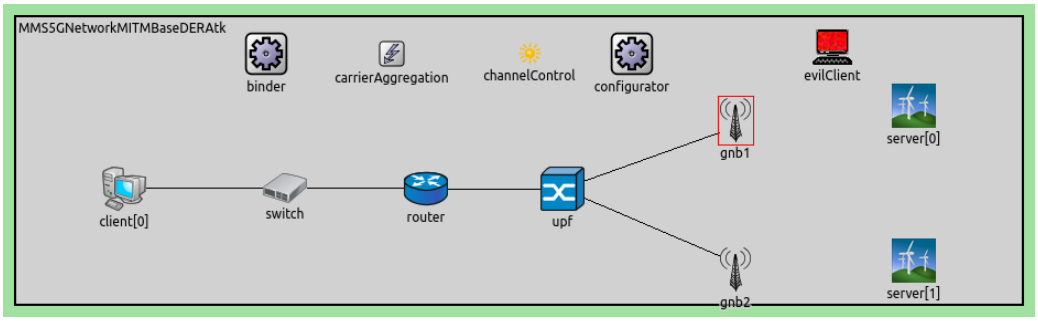


Figura 5.13: Rete `MMS5GNetworkMITMBaseDERAtk.ned`

Nel codice 5.39 troviamo la configurazione `MMS_MITM_5G_DER_Atk` che presenta alcune differenze rispetto a quelle precedenti. A parte l'indicazione della rete di riferimento nel parametro `network` (`MMS5GNetworkMITMBaseDERAtk`), possiamo notare che `app[0]` e `app[1]` del `client` sono connesse rispettivamente all'`evilClient` e al `server[1]` direttamente (attraverso il parametro `connectAddress`) siccome solo la connessione al `server[0]` è stata compromessa. Inoltre visto che l'`evilClient` è un `NRUe` dobbiamo impostare i parametri `macCellId`, `masterId`, `nrMacCellId` e `nrMasterId` per contenere l'id della cella a cui l'host si dovrà connettere (`gnb1`), proprio come fatto per i `server` nella configurazione estesa `MMS_MITM_5G_Cli_Atk_Base`.

```
1  [MMS_MITM_5G_DER_Atk]
2  extends = MMS_MITM_5G_Cli_Atk_Base
3  network = MMS5GNetworkMITMBaseDERAtk
4
5  *.evilClient.mobility.initFromDisplayString = true
6
7  # Specifies that the app[0] of the evilClient shall connect
       to the server[0] and the
8  # app[1] shall connect to the server[1].
9  *.evilClient.app[0].connectAddress = "
       MMS5GNetworkMITMBaseDERAtk.server[0]"
10 *.evilClient.app[1].connectAddress = "
       MMS5GNetworkMITMBaseDERAtk.server[1]"
11
```

```
12 # We simulate that the connection with the server 0 is
       compromised but
13 # the connection with the server 1 is direct without the
       evilClient in between
14 *.client[*].app[0].connectAddress = "
       MMS5GNetworkMITMBaseDERAtk.evilClient"
15 *.client[*].app[1].connectAddress = "
       MMS5GNetworkMITMBaseDERAtk.server[1]"
16
17 **.evilClient.macCellId = 1
18 **.evilClient.masterId = 1
19 **.evilClient.nrMacCellId = 1
20 **.evilClient.nrMasterId = 1
21
22 **.gnb*.gateway = "MMS5GNetworkMITMBaseDERAtk.upf"
```

Codice 5.39: Configurazione `MMS_MITM_5G_DER_Atk`

MMS_MITM_5G_DER_Atk_Delayed

In quest'ultima configurazione (la cui rete è visibile in Figura 5.14) abbiamo una topologia analoga a quella usata per `MMS_MITM_5G_DER_Atk` ma il `router` è stato sostituito con un `internetCloud` delayer per introdurre ritardi aggiuntivi.

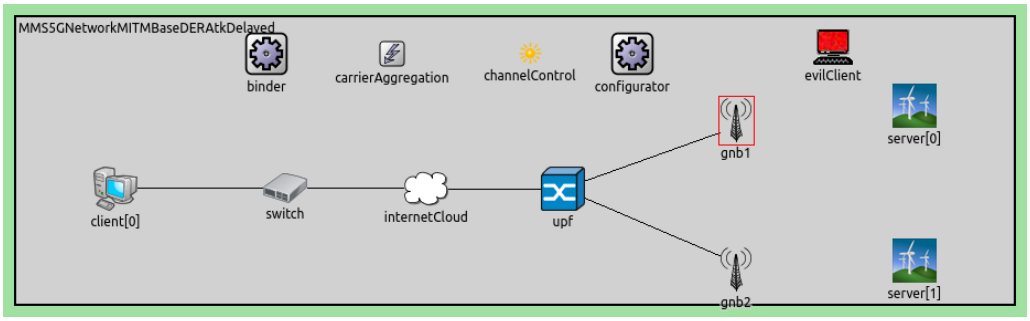


Figura 5.14: Rete `MMS5GNetworkMITMBaseDERAtkDelayed.ned`

Anche guardando alla rispettiva configurazione nel codice 5.40 non si notano evidenti cambiamenti rispetto a quella sopra citata, se non il riferimento alla nuova rete adottata nel parametro `network` e nella specifica degli indirizzi di connessione (`MMS5GNetworkMITMBaseDERAtkDelayed`).

```
1 [MMS_MITM_5G_DER_Atk_Delayed]
2 extends = MMS_MITM_5G_Cli_Atk_Base
3 network = MMS5GNetworkMITMBaseDERAtkDelayed
4
5 *.evilClient.mobility.initFromDisplayString = true
6
7 # Specifies that the app[0] of the evilClient shall connect
      to the server[0] and the
8 # app[1] shall connect to the server[1].
```

```
9  *.evilClient.app[0].connectAddress = "
       MMS5GNetworkMITMBaseDERAtkDelayed.server[0]"
10 *.evilClient.app[1].connectAddress = "
       MMS5GNetworkMITMBaseDERAtkDelayed.server[1]"
11
12 # We simulate that the connection with the server 0 is
       compromised but
13 # the connection with the server 1 is direct without the
       evilClient in between
14 *.client[*].app[0].connectAddress = "
       MMS5GNetworkMITMBaseDERAtkDelayed.evilClient"
15 *.client[*].app[1].connectAddress = "
       MMS5GNetworkMITMBaseDERAtkDelayed.server[1]"
16
17 **.evilClient.macCellId = 1
18 **.evilClient.masterId = 1
19 **.evilClient.nrMacCellId = 1
20 **.evilClient.nrMasterId = 1
21
22 **.gnb*.gateway = "MMS5GNetworkMITMBaseDERAtkDelayed.upf"
```

Codice 5.40: Configurazione `MMS_MITM_5G_DER_Atk_Delayed`

In Tabella 5.1 è visibile un riassunto delle configurazioni scritte per l'attacco di tipo Man In The Middle. In particolare sono elencate le caratteristiche salienti come il tipo di connessione del client, il tipo di connessione del server, l'entità compromessa e se è stato usato un `Router` o un `Delayer`. Non è stato incluso il tipo di connessione dell'attaccante in quanto corrispondente con il mezzo di comunicazione del modulo compromesso.

5.2.7 Le configurazioni realizzate: Distributed Denial Of Service

Passiamo ora alla presentazione delle configurazioni per l'attacco di Distributed Denial Of Service. Anche in questo caso la configurazione `General` viene estesa di default, ma non la commenteremo nuovamente per evitare ripetizioni.

MMS_DoS_Base

La `MMS_DoS_Base` è una configurazione di base che contiene parametri comuni a tutte quelle realizzate per questo tipo di attacco. Dopo aver impostato il delayer con l'apposito file "*.xml*", viene configurato il numero di `client` (1 normale e 10 malevoli) e quello di `server` (supponiamo siano 2). In particolare al primo `client` vengono associate due applicazioni per potersi collegare a entrambi i `server` DER, mentre per gli altri (quelli malevoli) ne basta solo una per collegarsi all'unico `server` sotto attacco (`server[0]`). In seguito viene effettuata la configurazione dei `server` impostando un `replyDelay` nullo, un `serviceTime` con distribuzione uniforme tra $30ms$ e $40ms$ (così da poter usare un numero di client compromessi non eccessivamente elevato) e un intervallo d'invio di misure periodiche di $4s$. Da riga 24 a riga 26 impostiamo la simulazione per avere un solo client normale, mentre i restanti

sono tutti client malevoli con un tempo d'invio tra una richiesta malformata e l'altra nell'ordine dei $50ms$. L'ultima riga della configurazione va a disabilitare gli eventi di timeout che potrebbero scattare visti i ritardi nella ricezione dei messaggi, ma che in questo tipo di simulazioni non ci interessano.

```
1  # MMS DoS Base configuration extended by every other MMS DoS
       configuration
2  [MMS_DoS_Base]
3
4  # Internet cloud delayer configuration
5  **.internetCloud.ipv4Delayer.config = xmldoc("
       internetCloudDelays.xml")
6
7  # Set the number of modules
8  *.numServers = 2
9  *.numGoodClient = 1
10 *.numBadClient = 10
11
12 # The normal client shall conect to both the servers
13 *.client[0].numApps = 2
14 *.client[1..].numApps = 1
15
16 # Server Settings
17 **.server*.app[*].typename = "MmsServer"
18 **.server*.app[*].replyDelay = 0ms
19 **.server*.app[*].serviceTime = intuniform(30ms, 40ms)
20 **.server*.app[*].emitInterval = 4s
21
22 # Client Settings (define which clients are good ones and
       which are bad ones
23 # with the respective service time distributions)
24 **.client[0].app[*].typename = "MmsClient"
25 **.client[1..].app[*].typename = "BadMmsClient"
26 **.client[1..].app[*].thinkTime = intuniform(47ms, 52ms)
27
28 ...
29
30 # It's better to disable all the timeouts
31 **.client[*].app[*].resTimeoutInterval = 120s
```

Codice 5.41: Configurazione `MMS_DoS_Base`

MMS_DoS_Wired

In questa prima configurazione DoS (visibile in Figura 5.15) abbiamo scelto di mantenere una rete completamente cablata dove tutti i `client` (compromessi e non) sono connessi a uno `switch`, il quale è collegato a un `internetCloud` delayer a cui sono connessi entrambi i `server` DER. Tutti i canali di comunicazione usati hanno una velocità di trasmissione di $100Mbps$, tranne quello tra `switch` e delayer

che è da 1*Gbps*. Come già visto precedentemente il modulo `configurator` svolge le procedure di configurazione e routing iniziali.

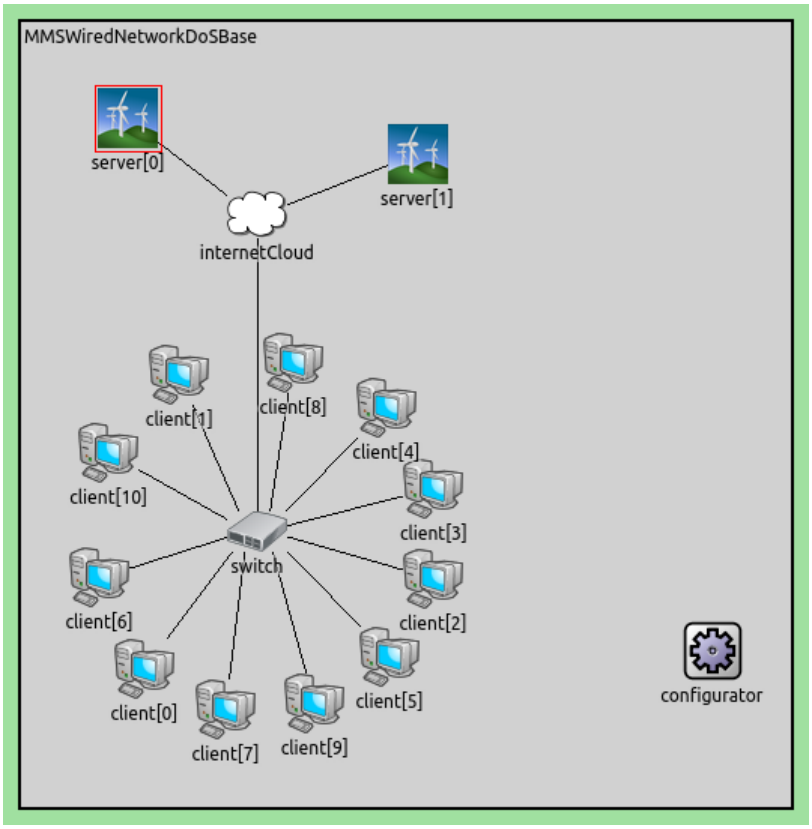


Figura 5.15: Rete `MMSWiredNetworkDoSBase.ned`

Nella rispettiva configurazione visibile nel codice 5.42, dopo la dichiarazione di estensione, la `app[0]` di tutti i `client` viene impostata per collegarsi al `server[0]`, mentre solo la `app[1]` del `client[0]` è configurata per collegarsi al `server[1]`.

```
1 [MMS_DoS_Wired]
2 network = MMSWiredNetworkDoSBase
3 extends = MMS_DoS_Base
4
5 # All the clients are connected to the server
6 *.client[*].app[0].connectAddress = "MMSWiredNetworkDoSBase.
     server[0]"
7 *.client[0].app[1].connectAddress = "MMSWiredNetworkDoSBase.
     server[1]"
```

Codice 5.42: Configurazione `MMS_DoS_Wired`

MMS_DoS_Wireless

In questo tipo di configurazione tutti gli host nella rete sono dei `WirelessHost` con due `accessPoint` (uno per i `client` e uno per i `server`) connessi a un

`internetCloud` delayer tramite canale Ethernet a 1*Gbps*. Tra i vari moduli possiamo trovare anche un `radioMedium` per la rappresentazione del canale di comunicazione, un `visualizer` per la visualizzazione degli scambi wireless e il classico `configurator` presente in tutte le reti.

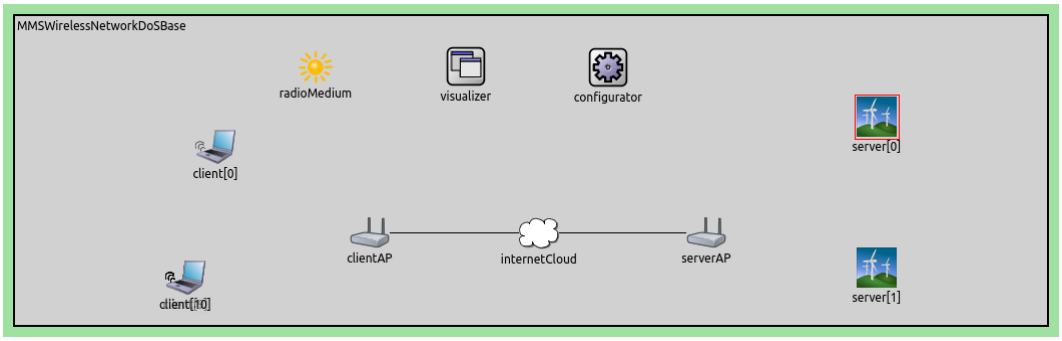


Figura 5.16: Rete `MMSWirelessNetworkDoSBase.ned`

Nell'estratto di configurazione visibile nel codice 5.43 possiamo trovare immediatamente l'associazione del `connectAddress` di ciascuna `app` all'interno dei client al corrispettivo server (come già visto nella variante cablata). In seguito andiamo a spegnere la registrazione dei vettori di misure per i client compromessi, così da evitare di generare inutilmente dei file eccessivamente grandi. Nelle righe che vanno dalla 14 alla 30 troviamo la configurazione della posizione statica (`StationaryMobility`) per i vari `client` e `server`, così che ogni elemento dei diversi vettori sia graficamente distinguibile dagli altri (purtroppo per limiti del sistema di posizionamento i vari client compromessi risultano sovrapposti). Nell'ultima parte della configurazione sono presenti i comandi per inizializzare staticamente gli SSID nei diversi `accessPoint` (si agisce sulla scheda wireless), in modo da associare ciascun `client` e `server` al punto d'accesso prestabilito.

```
1  # MMS DoS configuration on Wireless network with just the
       clients, multiple servers and an Access Point
2  [MMS_DoS_Wireless]
3  network = MMSWirelessNetworkDoSBase
4  extends = MMS_DoS_Base
5
6  # The clients are connected to each server
7  *.client[*].app[0].connectAddress = "
       MMSWirelessNetworkDoSBase.server[0]"
8  *.client[0].app[1].connectAddress = "
       MMSWirelessNetworkDoSBase.server[1]"
9
10 # Disable vector recording for the compromised clients
11 *.client[1..].**.vector-recording = false
12
13 # Fixed position of the clients
14 *.client[*].mobility.typename = "StationaryMobility"
15 *.server[*].mobility.typename = "StationaryMobility"
16 # Set to false if you want to use stationary mobility
```

```
17 *.client[*].mobility.initFromDisplayString = false
18 *.server[*].mobility.initFromDisplayString = false
19
20
21 *.client[*].mobility.initialZ = 0m
22 *.server[*].mobility.initialZ = 0m
23 *.client[0].mobility.initialX = 200m
24 *.client[0].mobility.initialY = 130m
25 *.client[1..].mobility.initialX = 170m
26 *.client[1..].mobility.initialY = 260m
27
28 *.server[*].mobility.initialX = 860m
29 *.server[0].mobility.initialY = 100m
30 *.server[1].mobility.initialY = 250m
31
32
33 # Statically binding client and server to the respective APs
34 **.clientAP.wlan[*].mgmt.ssid = "cliAP"
35 **.serverAP.wlan[*].mgmt.ssid = "serAP"
36 **.client[*].wlan[*].agent.defaultSsid = "cliAP"
37 **.server[*].wlan[*].agent.defaultSsid = "serAP"
```

Codice 5.43: Configurazione `MMS_DoS_Wireless`

MMS_DoS_Wired_5G

In quest'ultima configurazione abbiamo adottato una topologia simile a quella usata per la `MMS_DoS_Wired`, ma in questo caso i server sono degli `NRUe` collegati tramite rete 5G a delle `gNodeBs` a loro volta connesse alla rete principale attraverso un `upf` centrale. Tra gli altri troviamo anche i moduli globali necessari per il funzionamento della rete 5G, di cui abbiamo già discusso precedentemente.

Nella configurazione visibile nel codice 5.44, dopo la dichiarazione di estensione sono presenti le inizializzazioni dei parametri `connectAddress` dei client e l'inserimento delle posizioni statiche per i due server presenti. Infine da riga 15 a riga 25 è presente l'associazione tra `server[0]` e `server[1]` rispettivamente alla `gnb1` e alla `gnb2` per mezzo dei loro identificatori.

```
1  [MMS_DoS_Wired_5G]
2  network = MMSWired5GNetworkDoSBase
3  extends = MMS_DoS_Base
4
5  # All the clients are connected to the server
6  *.client[*].app[0].connectAddress = "
       MMSWired5GNetworkDoSBase.server[0]"
7  *.client[0].app[1].connectAddress = "
       MMSWired5GNetworkDoSBase.server[1]"
8
9  **.server[*].mobility.initFromDisplayString = false
10 **.server[*].mobility.initialY = 40m
```

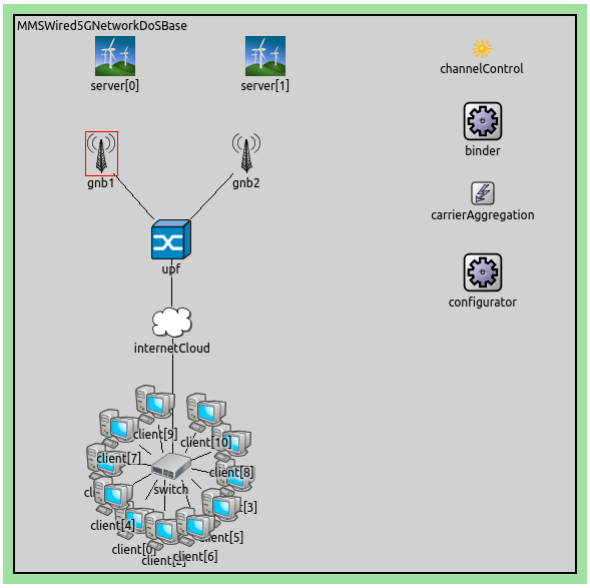


Figura 5.17: Rete `MMSWired5GNetworkDoSBase.ned`

```
**.server[0].mobility.initialX = 100m
**.server[1].mobility.initialX = 250m

# Connect the NRUE's NIC to the corresponding serving gNB
**.server[0].macCellId = 1
**.server[0].masterId = 1
**.server[0].nrMacCellId = 1
**.server[0].nrMasterId = 1

**.server[1].macCellId = 2
**.server[1].masterId = 2
**.server[1].nrMacCellId = 2
**.server[1].nrMasterId = 2

**.gnb*.gateway = "MMSWired5GNetworkDoSBase.upf"
```

Codice 5.44: Configurazione `MMS_DoS_Wired_5G`

Nella Tabella 5.2 possiamo vedere un riassunto delle configurazioni realizzate per l'attacco di Distributed Denial Of Service, dove sono indicati anche i tipi di connessioni di `client` e `server`.

Nome	*Fig.*	*Rete Client*	*Rete Server*	*Modulo attaccato*	*Router o Delayer*
MMS_MITM _Wired_Cli_Atk	5.3	Ethernet	Ethernet	`client[0]`	Router
MMS_MITM _Wired_Cli _Atk_Delayed	5.4	Ethernet	Ethernet	`client[0]`	Delayer
MMS_MITM _Wired_DER_Atk	5.5	Ethernet	Ethernet	`server[0]`	Router
MMS_MITM _Wired_DER _Atk_Delayed	5.6	Ethernet	Ethernet	`server[0]`	Delayer
MMS_MITM _Wireless_Cli_Atk	5.7	Wireless	Ethernet	`client[0]`	Router
MMS_MITM _Wireless_Cli _Atk_Delayed	5.8	Wireless	Ethernet	`client[0]`	Delayer
MMS_MITM _Wireless_DER _Atk	5.9	Ethernet	Wireless	`server[0]`	Router
MMS_MITM _Wireless_DER _Atk_Delayed	5.10	Ethernet	Wireless	`server[0]`	Delayer
MMS_MITM _5G_Cli_Atk	5.11	Ethernet	5G	`client[0]`	Router
MMS_MITM _5G_Cli _Atk_Delayed	5.12	Ethernet	5G	`client[0]`	Delayer
MMS_MITM _5G_DER_Atk	5.13	Ethernet	5G	`server[0]`	Router
MMS_MITM _5G_DER _Atk_Delayed	5.14	Ethernet	5G	`server[0]`	Delayer

Tabella 5.1: Tabella riassuntiva delle configurazioni per l'attacco MITM

Nome	*Fig.*	*Rete Client*	*Rete Server*
MMS_DoS_Wired	5.15	Ethernet	Ethernet
MMS_DoS_Wireless	5.16	Wireless	Wireless
MMS_DoS_Wired_5G	5.17	Ethernet	5G

Tabella 5.2: Tabella riassuntiva delle configurazioni per l'attacco DDoS

Capitolo 6

Analisi dei risultati

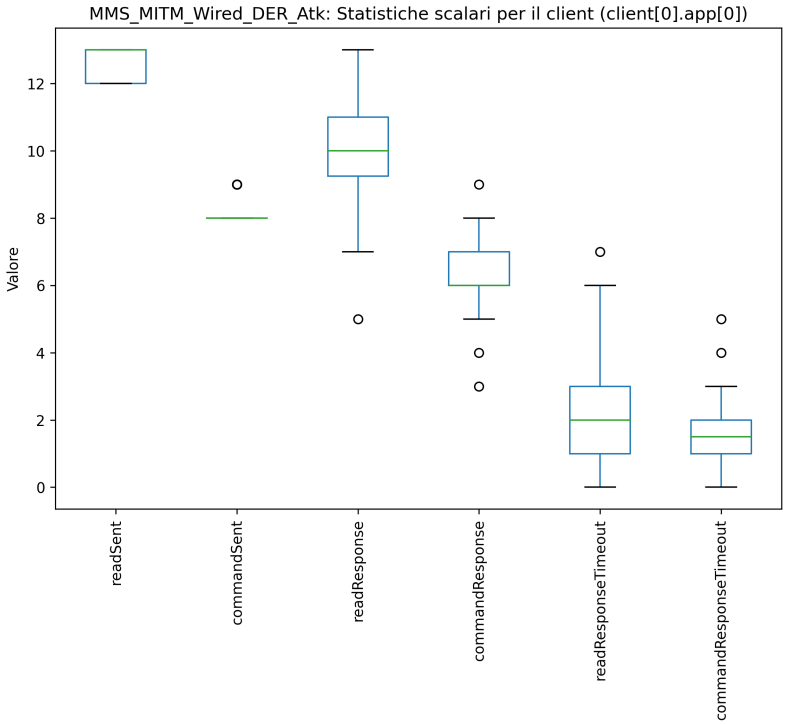
L'ultimo elemento che manca al nostro studio è l'**analisi delle misure** rilevate nelle diverse configurazioni. Abbiamo già visto come direttamente all'interno dell'IDE sia possibile visionare le misure ottenute nelle singole run, indipendentemente che si tratti di scalari o vettori di valori. Ciò che non è direttamente supportato è l'analisi delle distribuzioni di tali misure a partire da un campione ottenuto eseguendo dei batch di simulazioni, ciascuno con un insieme di semi iniziali diverso. Per analizzare questo aspetto, a partire dalla versione 6.0 di *OMNeT++*, è disponibile una libreria Python (*Scave*) per la lettura agevolata di file di misure prodotti in fase di simulazione. Questo permette di evitare il passaggio di conversione dei file scalari (".*sca*") e vettoriali (".*vec*") nelle corrispettive versioni testuali di tipo *Comma-separated values* (".*csv*") da elaborare con sistemi esterni. In aggiunta, invece di caricare in memoria il dataset completo, è possibile creare delle *espressioni di filtro* che se passate ai metodi forniti dalla libreria possono ridurre l'insieme di misure inizialmente caricate a quelle che veramente servono in funzione dell'obiettivo delle simulazioni. Abbiamo già presentato in sezione 3.5 i diversi tipi di stimatori e di grafici che si è deciso di adottare, ma non sarà presentato nel dettaglio il codice Python prodotto per l'analisi. Nelle sezioni che seguiranno non saranno presentati tutti i grafici generati per ogni diversa configurazione, ma ci concentreremo sugli aspetti principali che caratterizzano una certa simulazione e la differenziano dalle altre. Citeremo nuovamente i valori dei parametri che influiscono maggiormente su una data statistica, mentre per le configurazioni complete si può fare riferimento alle sezioni 5.2.6 e 5.2.7. In tutti i casi le simulazioni sono state eseguite per 50 ripetizioni, ognuna con un insieme di semi iniziali differente.

6.1 L'attacco Man In The Middle

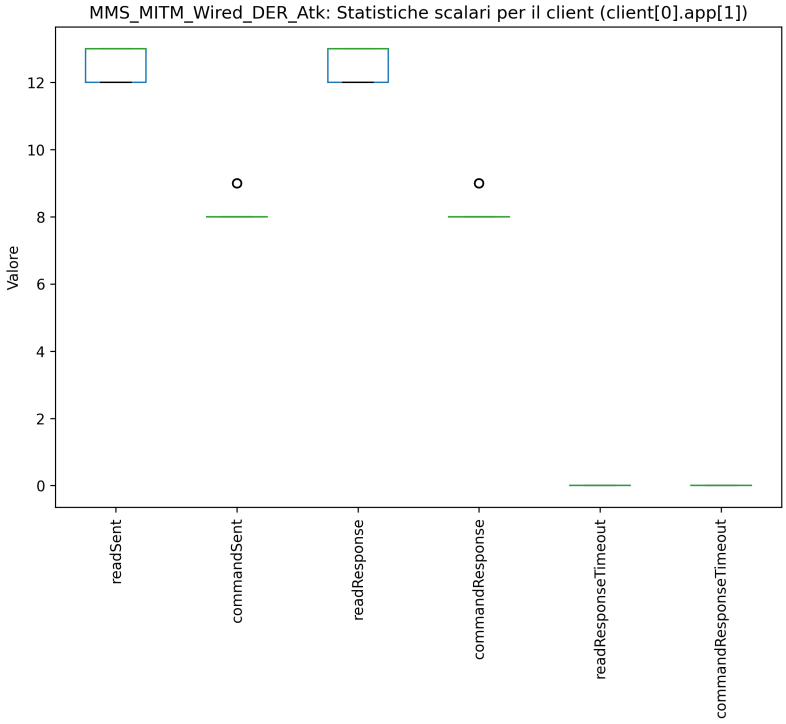
In base alle analisi svolte sulle diverse configurazioni considerate per l'attacco Man In The Middle si è rilevato che i fattori principali che influenzano le misure in output sono i seguenti: **posizionamento dell'attaccante**, **presenza del delayer** e **tecnologia di connessione**

Posizionamento dell'attaccante
A seconda che l'attaccante fosse posizionato nella sottorete del `client` o in quella di

uno dei `server` abbiamo modellato il fatto che entrambe o una sola delle connessioni client-server fossero compromesse.



(a) Connessione compromessa con il `server[0]`



(b) Connessione con il `server[1]`

Figura 6.1: Misure scalari del `client`

Nome	***Assegnazione***
sendReadInterval (`client`)	`intuniform(8s, 10s)`
sendCommandInterval (`client`)	`intuniform(12s, 15s)`
thinkTime (`evilClient`)	`intuniform(5ms, 8ms)`
replyDelay (`evilClient`)	`intuniform(0ms, 5ms)`
serviceTime (`server`)	`intuniform(2ms, 4ms)`
emitInterval (`server`)	`4s`

Tabella 6.1: Parametri principali usati per le diverse configurazioni

Nome	***AVG***	***CI 98%***	***RA %***
readSent	12.7	±0.136	1.07
commandSent	8.1	±0.086	1.06
readResponse	10.0	±0.437	4.39
commandResponse	6.4	±0.329	5.17
readResponseTimeout	2.3	±0.435	18.76
commandResponseTimeout	1.6	±0.314	19.93

Tabella 6.2: Media, intervallo di confidenza al 98% e accuratezza relativa delle misure scalari del `client` nella connessione con il `server[0]` (configurazione MMS_MITM_Wired_DER_Atk)

Come possiamo vedere nelle Figure 6.1a e 6.1b sono presenti i box plot relativi alle misure scalari associate alle connessioni con il `server[0]` e il `server[1]`. Nella Tabella 6.1 sono elencati i parametri principali adottati in questa e nelle altre configurazioni. Visto che in questa configurazione solo una delle due connessioni era compromessa (la prima), possiamo notare alcune differenze tra i due grafici. Mentre le statistiche riguardanti le richieste di lettura di misure e d'esecuzione di comandi si equivalgono tra le due connessioni, ciò che varia consistentemente è il numero di risposte ricevute. Nella connessione con il `server[1]` tutte le risposte vengono ricevute correttamente e se ne può avere conferma dal fatto che le misure relative al numero di risposte in timeout (`readResponseTimeout` e `commandResponseTimeout`) sono sempre a 0. Invece per la connessione con il `server[0]` è facile notare come alcune richieste o risposte vengano bloccate dall'attaccante (la deviazione standard di `readResponse` e `commandResponse` è maggiore e il valore medio è sempre più basso nella connessione compromessa). Riceviamo una doppia conferma di questo fenomeno anche guardando alla distribuzione non nulla di `readResponseTimeout` (valore medio 2.3 ± 0.435) e `commandResponseTimeout` (valore medio 1.6 ± 0.314). Per un riferimento più preciso riguardo alle statistiche appena citate vedere le Tabelle 6.2 e 6.3 che contengono medie, intervalli di confidenza e accuratezze relative di ogni scalare misurato.

Se guardiamo alla Figura 6.2 possiamo osservare il numero di misure periodiche ricevute ogni 4 secondi dal `server[0]`. Siccome tale intervallo è identico al periodo

Nome	***AVG***	***CI 98%***	***RA %***
readSent	12.6	±0.139	1.10
commandSent	8.2	±0.123	1.49
readResponse	12.6	±0.139	1.10
commandResponse	8.2	±0.123	1.49
readResponseTimeout	0	±0	-
commandResponseTimeout	0	±0	-

Tabella 6.3: Media, intervallo di confidenza al 98% e accuratezza relativa delle misure scalari del `client` nella connessione con il `server[1]` (configurazione MMS_MITM_Wired_DER_Atk)

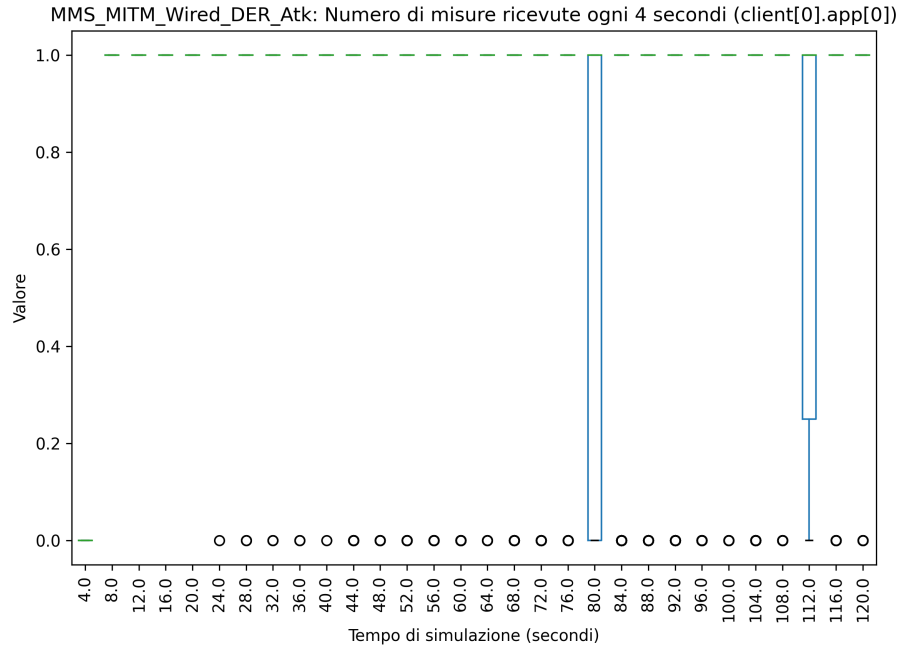


Figura 6.2: Misure periodiche ricevute dal `client` sulla connessione compromessa con il `server[0]`

d'invio delle misure, se la connessione non fosse compromessa dovremmo registrare il valore 1 in ogni lasso temporale. Come invece possiamo notare grazie agli *outliers* nel grafico, in alcune run non viene ricevuto alcun messaggio periodico per alcuni lassi di tempo (specialmente gli ultimi dove l'`evilClient` sta sferrando l'attacco con potenza massima). Una visione complessiva dal punto di vista dell'attaccante ci viene data dalle misure da esso generate che possiamo vedere riassunte in Figura 6.3 e dalla rispettiva Tabella 6.4. Ciò che emerge da questi due elementi evidenzia eventuali sbilanciamenti nel blocco o nella compromissione di richieste e risposte relative a variabili o comandi. Se ad esempio l'attaccante tendesse a prediligere il blocco di richieste di lettura di variabili (`readRequestBlock`) rispetto alle risposte (`readResponseBlock`), non potremmo osservarlo direttamente dalle misure

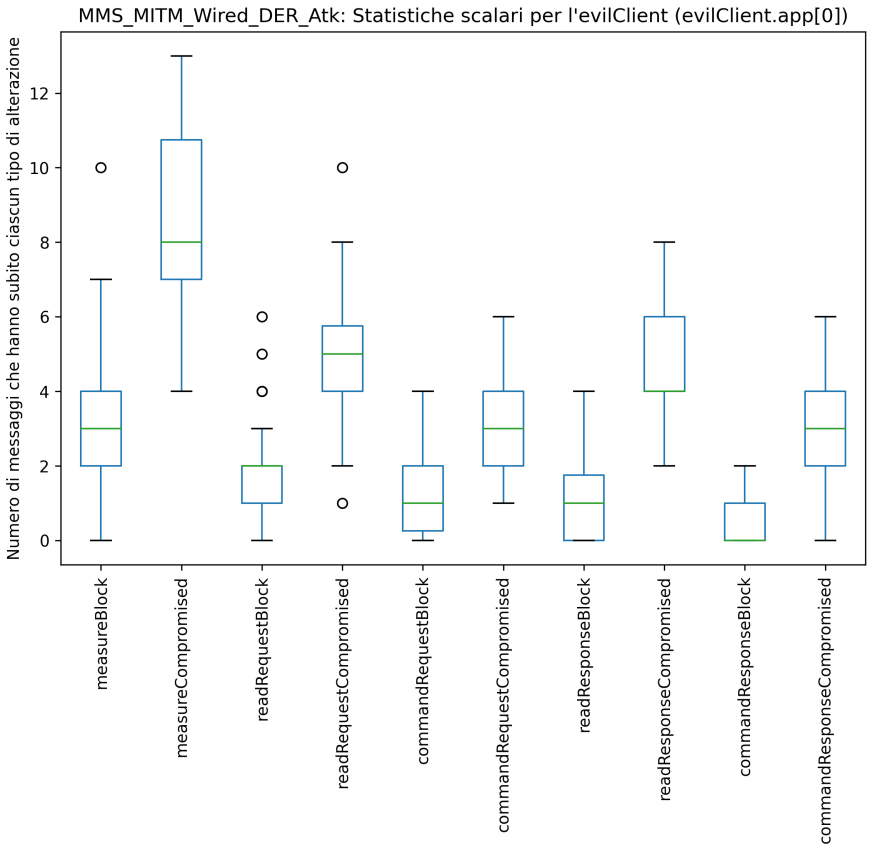


Figura 6.3: Misure scalari dell'`evilClient` relative alla connessione compromessa con il `server[0]`

del `client` compromesso, per cui in ogni caso otterremmo un timeout. I risultati ottenuti tramite queste misure dipendono dalle probabilità a priori configurate sull'attaccante, dal tasso d'invio dei messaggi sia da parte del `client` che da parte del `server` e dai valori d'inibizione determinati dallo stato della FSM in cui si trova l'entità malevola.
Un altro fenomeno causato dalla presenza di una connessione alterata è l'incremento del tempo di risposta alle richieste MMS. In particolare possiamo confrontare le Figure 6.4 e 6.5, dove nel primo caso la connessione è compromessa, mentre nel secondo caso non lo è. Queste statistiche sono state suddivise in range temporali e non sulla base dei singoli messaggi per permetterci di aggregare run diverse dove non è detto che vi sia una corrispondenza temporale nella ricezione delle varie risposte. In questo modo, dato il numero d'intervalli da generare, possiamo valutare il tempo di risposta in ogni fase della simulazione. Ciò che emerge dalle due immagini citate è che l'operato dell'attaccante introduce un certo ritardo nelle risposte. Mentre in una situazione normale i tempi di risposta alle singole richieste di misure hanno valore medio di $3.8ms$ e intervallo di confidenza al 98% di $\pm 0.157ms$, in una situazione di attacco MITM questi valori si alzano fino ad avere valore medio di $20.7ms$ con un intervallo di confidenza al 98% di $\pm 0.698ms$. Ovviamente questa latenza aggiuntiva può essere smorzata o incrementata sulla base di come andiamo a configurare il tempo di servizio dell'attaccante.

Nome	***AVG***	***CI 98%***	***RA %***
measureBlock	3.0	±0.473	15.57
measureCompromised	8.6	±0.705	8.16
readRequestBlock	1.7	±0.354	20.59
readRequestCompromised	4.7	±0.0.470	10.00
commandRequestBlock	1.3	±0.319	25.29
commandRequestCompromised	3.2	±0.391	12.13
readResponseBlock	1.0	±0.260	26.52
readResponseCompromised	4.7	±0.473	10.03
commandResponseBlock	0.5	±0.184	38.28
commandResponseCompromised	2.8	±0.464	16.34

Tabella 6.4: Media, intervallo di confidenza al 98% e accuratezza relativa delle misure scalari dell'`evilClient` nella connessione compromessa con il `server[0]`(configurazione MMS_MITM_Wired_DER_Atk)

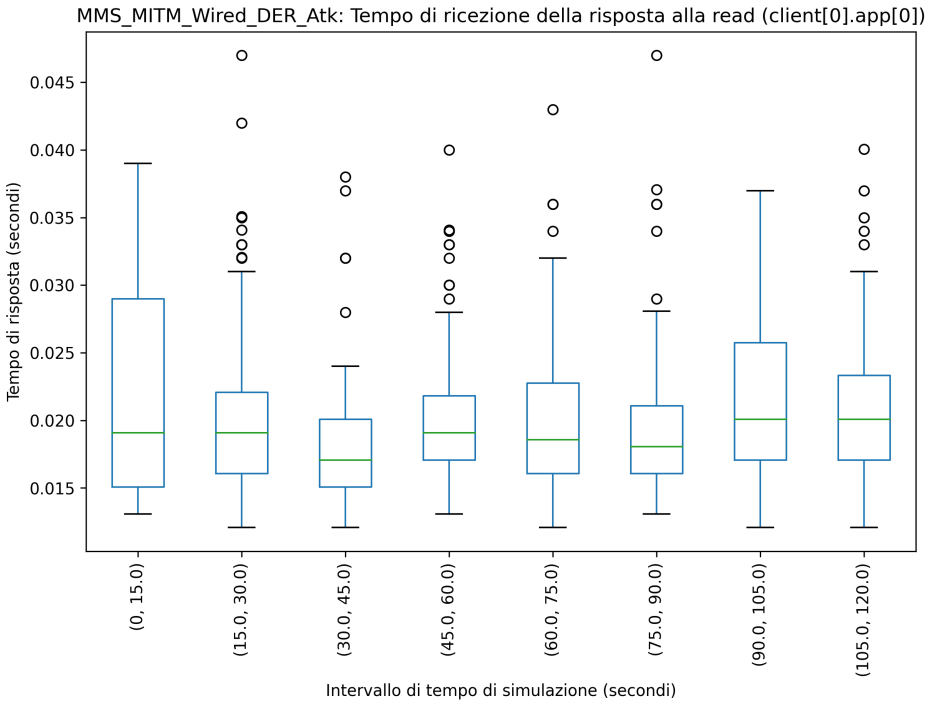


Figura 6.4: Tempo di risposta alle richieste di lettura di misura con il `server[0]`

Presenza del delayer

Nelle configurazioni in cui è stato introdotto un `InternetCloud` delayer tutti i messaggi che lo attraversano subiscono un ritardo sulla base delle impostazioni con cui è stato inizializzato. Guardando alle Figure 6.5 e 6.6 troviamo i box plot dei tempi di risposta tra `client` e `server[1]` (connessione non compromessa) senza

e con il delayer nella topologia. Analizzando la situazione senza delayer notiamo come il tempo di risposta medio sia di $3.8ms$ con un intervallo di confidenza al 98% di $\pm 0.157ms$, invece nell'altro caso il valore medio è di $185.2ms$ con un intervallo di confidenza al 98% di $\pm 3.68ms$, con anche diversi *outliers* che posseggono valori ben superiori.

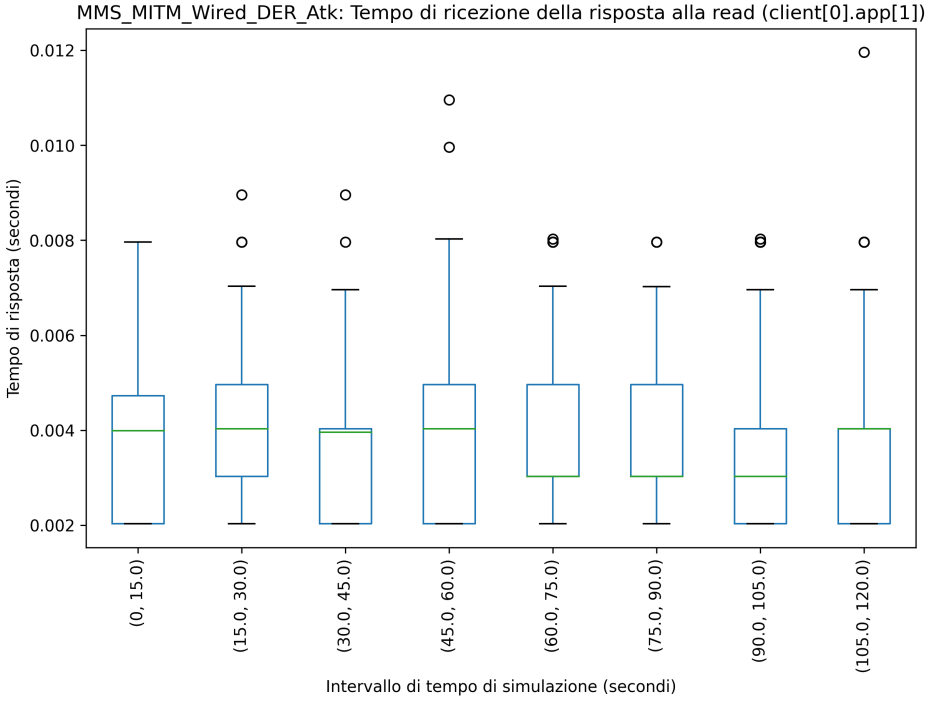


Figura 6.5: Tempo di risposta alle richieste di lettura di misura con il `server[1]` senza delayer

Tecnologia di connessione

L'ultimo aspetto da tenere in considerazione nella valutazione delle diverse simulazioni è la tecnologia di connessione usata dagli host. Per poter studiare meglio questa variabile abbiamo scelto di mantenere stabile il mezzo di trasmissione usato dai `client` (Ethernet), facendo variare solo quello adottato dai `server` (Ethernet, Wireless e 5G). I grafici usati per evidenziare le differenze nei diversi scenari sono degli *scatter plot* contenenti i tempi di risposta alle richieste di lettura di variabile, in cui ciascun punto fa riferimento a una singola richiesta. Guardando alla configurazione con i `server` su connessione Ethernet (in Figura 6.7) e quella in cui essi sono collegati su rete Wi-Fi (in Figura 6.8), non vi sono grosse differenze evidenziate dalle medie complessive ($3.8ms$ contro $3.6ms$) e dagli intervalli di confidenza al 98% ($\pm 0.157ms$ contro $\pm 0.111ms$). Sembra però che la connessione Wireless introduca un'instabilità maggiore visto che sono meno evidenti gli allineamenti di punti determinati dal tempo di risposta del `server` (tale distribuzione di probabilità uniforme restituisce valori interi tra $2ms$ e $4ms$), ben visibili invece nella variante su rete Ethernet. Passando invece al caso in cui i server sono connessi su rete 5G (visibile in Figura 6.9), nonostante a un primo sguardo può sembrare che anche qui sia presente un allineamento dei valori, in realtà è un'illusione causata da un incremento

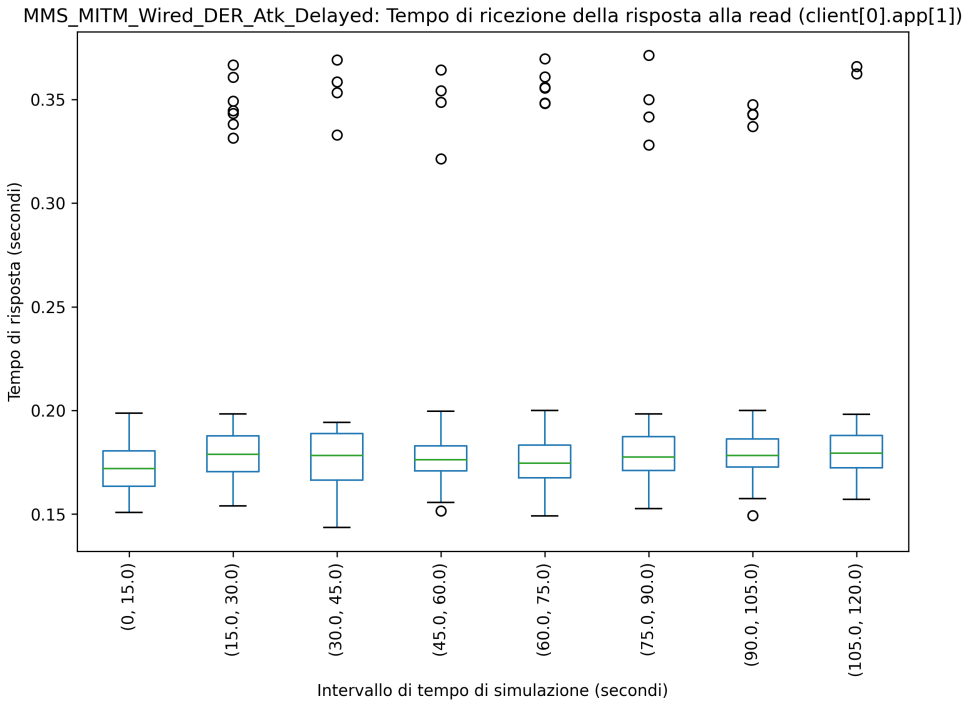


Figura 6.6: Tempo di risposta alle richieste di lettura di misura con il `server[1]` con delayer

notevole del tempo di risposta (valore medio $14.9ms$ e intervallo di confidenza al 98% di $\pm 0.321ms$) principalmente dato dal modello di trasmissione dei dati della connessione 5G.

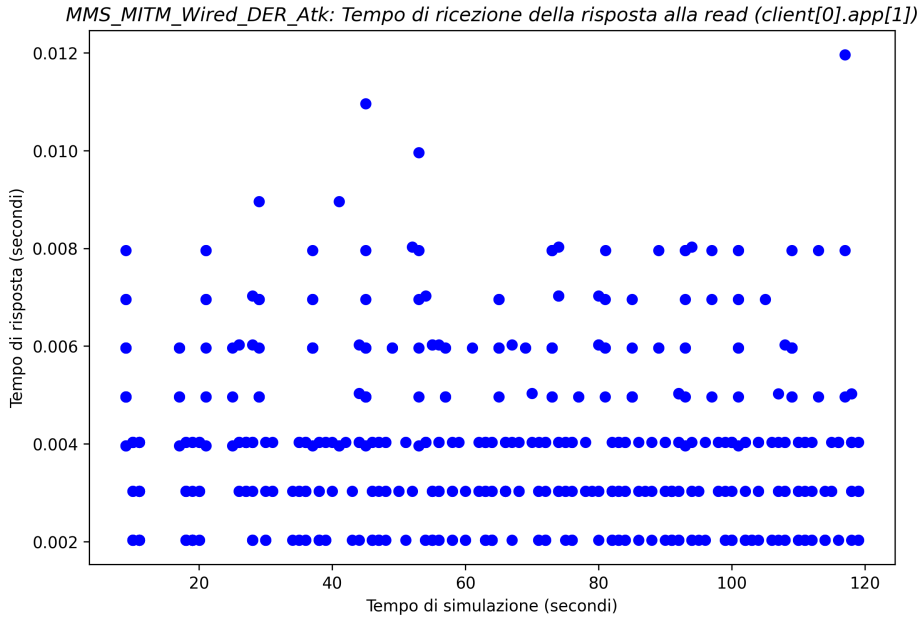


Figura 6.7: Scatter plot del tempo di risposta alle richieste di lettura di misura con il `server[1]` su rete Ethernet

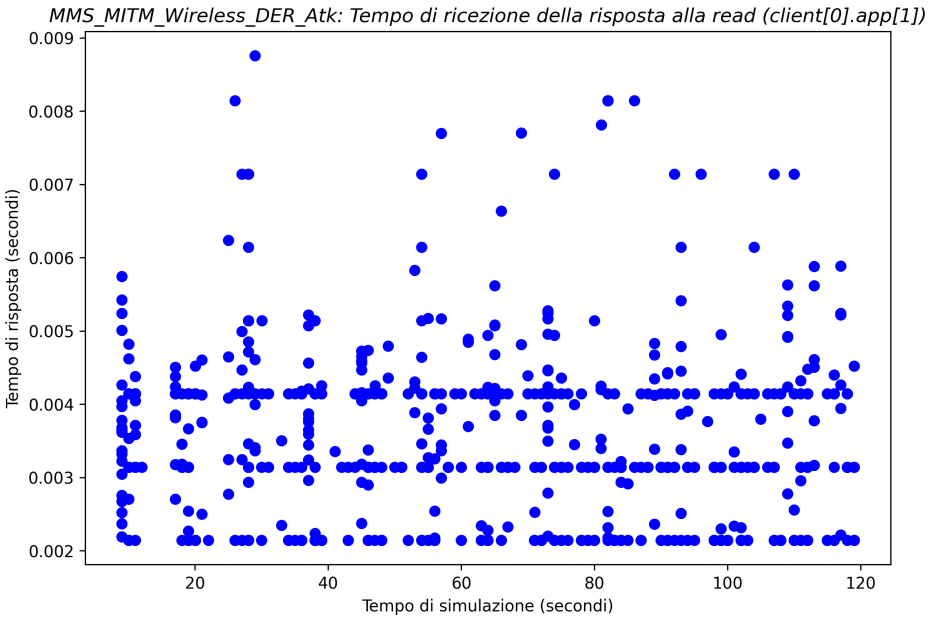


Figura 6.8: Scatter plot del tempo di risposta alle richieste di lettura di misura con il `server[1]` su rete Wireless

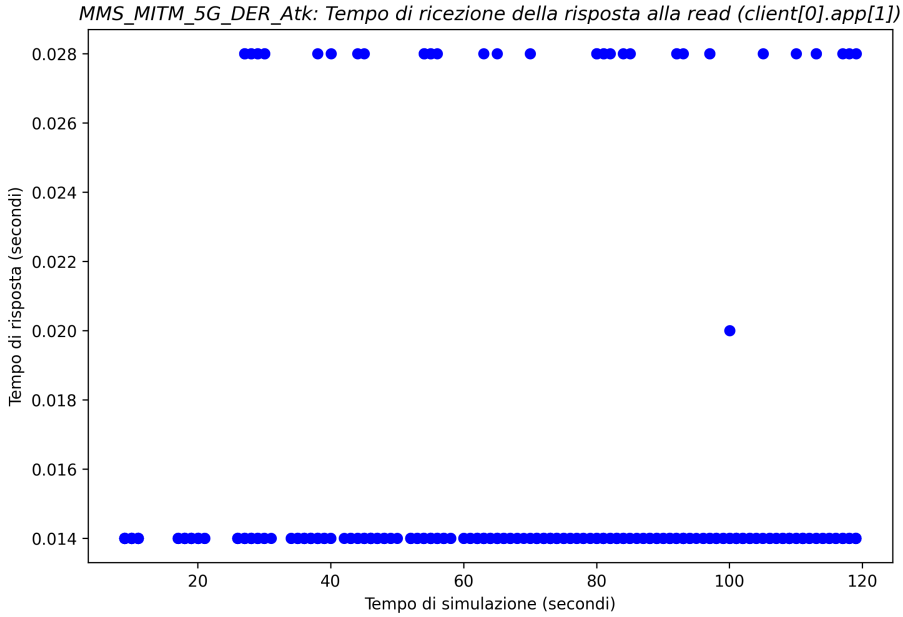


Figura 6.9: Scatter plot del tempo di risposta alle richieste di lettura di misura con il `server[1]` su rete 5G

6.2 L'attacco Distributed Denial of Service

Riguardo alle configurazioni che modellano l'attacco di Distributed Denial of Service abbiamo scelto d'incrementare il tempo di servizio dei `server` usando una distri-

buzione uniforme tra $30ms$ e $40ms$ in modo da poter usare soltanto 10 `client` malevoli (che iniziano immediatamente la loro attività) per riuscire a osservare i fenomeni caratteristici di questo tipo di attacchi, pur mantenendo la simulazione relativamente efficiente come tempo d'esecuzione. In ogni configurazione è sempre presente un `InternetCloud` delayer in modo da simulare più fedelmente i ritardi di una rete reale e in ogni caso abbiamo già osservato le differenze tra connessioni Ethernet, Wi-Fi e 5G che vengono invece mascherate proprio dal delayer. In tutte le simulazioni abbiamo scelto di far comunicare l'unico `client` normale con entrambi `server` presenti nella rete, ma soltanto il `server[0]` viene bersagliato dai `client` malevoli. In questo modo possiamo osservare le differenze che sussistono tra le due connessioni. Già guardando alle statistiche scalari mostrate in Figura 6.10 e nella Tabella 6.5 per la connessione con il `server[0]` possiamo notare come non vi sia una corrispondenza tra il numero di richieste inviate e delle risposte ricevute (di entrambe le tipologie presenti). Ciò è sintomo del fatto che il server sia sovraccaricato di richieste da parte dei client malevoli e non riesca a gestire quelle reali entro la fine della simulazione. Le statistiche di timeout (`readResponseTimeout` e `commandResponseTimeout`) sono sempre a 0 siccome abbiamo disabilitato i rispettivi segnali per evitare falsi positivi relativi a richieste soltanto in ritardo. La controprova di queste osservazioni ci viene data dalla Figura 6.11 e dalla Tabella 6.6 in cui valutando le statistiche relative alla connessione con il `server[1]` (quello non attaccato) possiamo vedere la corrispondenza tra i valori richieste e risposte di ciascun tipo.

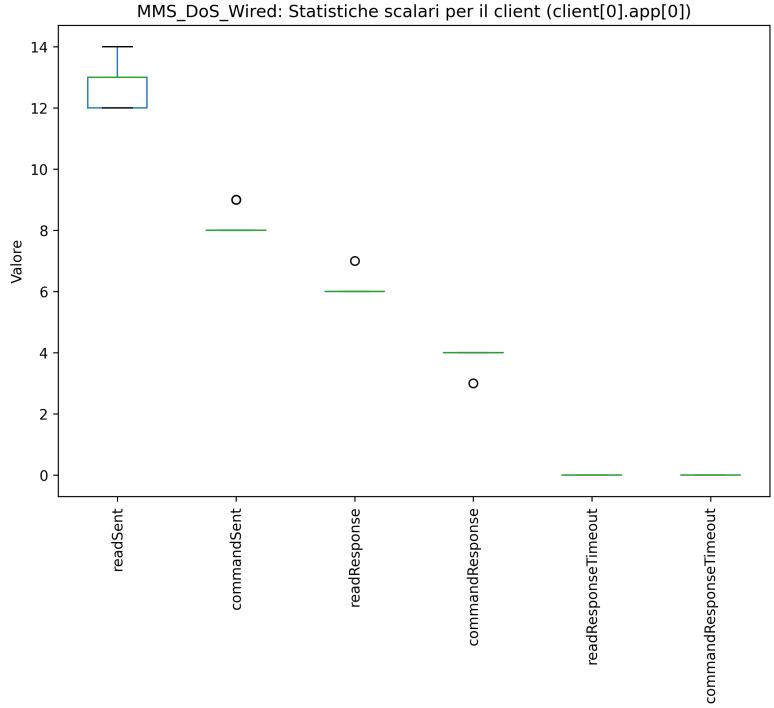


Figura 6.10: Misure scalari del `client` relative alla connessione compromessa con il `server[0]`

Anche la ricezione di misure periodiche da parte del `server[0]` è rallentata come evidenziato dal grafico in Figura 6.12, dove in alcuni intervalli di 4 secondi non

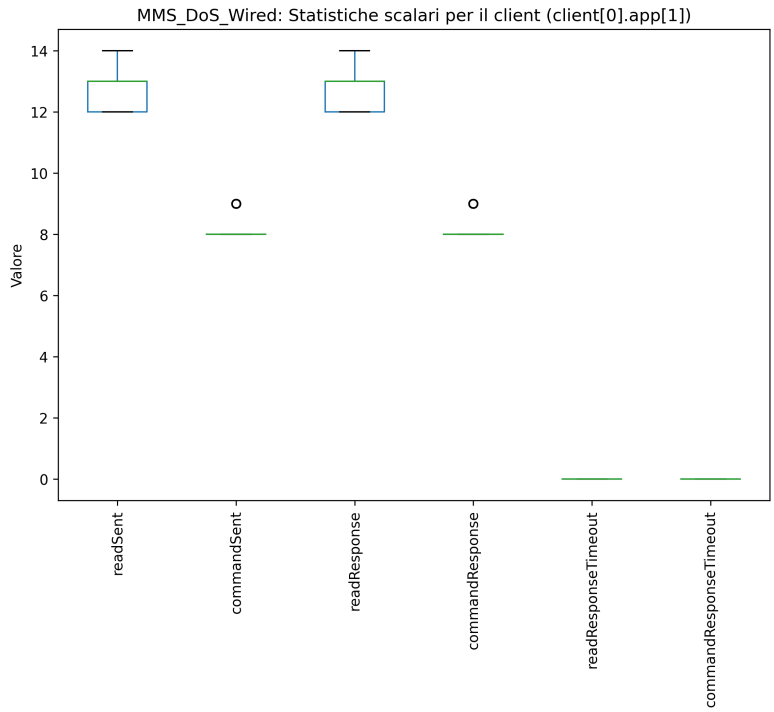


Figura 6.11: Misure scalari del `client` relative alla connessione compromessa con il `server[1]`

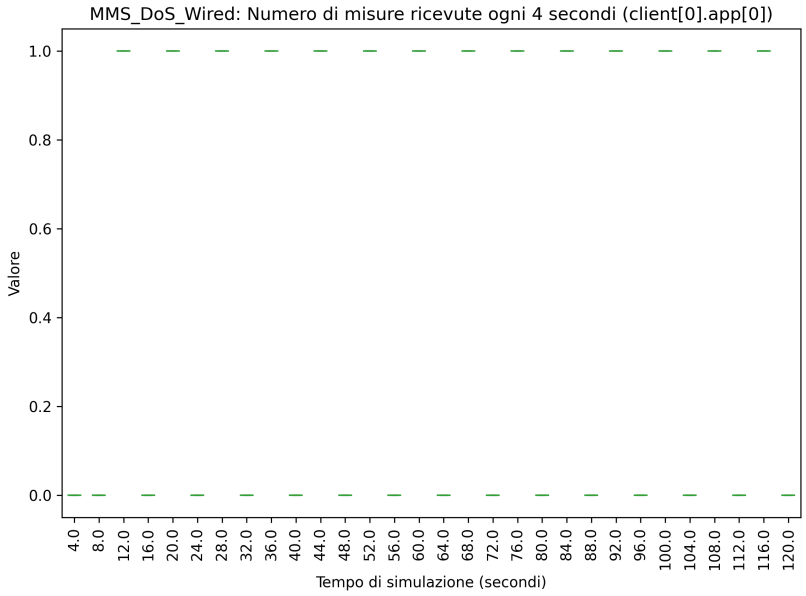


Figura 6.12: Misure periodiche ricevute dal `client` sulla connessione con il `server[0]`

viene ricevuta mai alcuna misura proprio perché ritardate dalla gestione di messaggi falsi da parte del server. Da notare è la differenza con il grafico in Figura 6.2, dove invece di un ritardo sistematico delle misure si verificava un blocco saltuario

Nome	***AVG***	***CI 98%***	***RA %***
readSent	12.7	±0.143	1.13
commandSent	8.1	±0.068	0.85
readResponse	6.0	±0.040	0.67
commandResponse	3.98	±0.040	1.01

Tabella 6.5: Media, intervallo di confidenza al 98% e accuratezza relativa delle misure scalari del `client` nella connessione con il `server[0]` (configurazione MMS_DoS_Wired)

Nome	***AVG***	***CI 98%***	***RA %***
readSent	12.8	±0.147	1.15
commandSent	8.2	±0.119	1.45
readResponse	12.8	±0.147	1.15
commandResponse	8.2	±0.119	1.45

Tabella 6.6: Media, intervallo di confidenza al 98% e accuratezza relativa delle misure scalari del `client` nella connessione con il `server[1]` (configurazione MMS_DoS_Wired)

in alcune run di simulazione, generando degli *outliers*. Al contrario se osservassimo lo stesso grafico relativo alla connessione con il `server[1]` (non compromesso) potremmo vedere che in ogni range viene sempre ricevuta una e una sola misura periodica.
L'ultimo aspetto che possiamo evidenziare in queste simulazioni di attacco di Distributed Denial of Service è l'incremento progressivo dei tempi di risposta da parte del server bersagliato. In questo caso abbiamo scelto di usare dei box plot in modo da far emergere le distribuzioni dei tempi di ricezione in ciascun range temporale in cui abbiamo suddiviso la simulazione (in questo caso ne abbiamo 8 da 15 secondi ciascuno). Dalla Figura 6.13 è facile notare come con il progredire della simulazione il tempo di risposta medio del `server[0]` aumenti in ciascun range fino al punto che alcune risposte non vengono mai ricevute (il valore medio complessivo è di $31.888s$ con un intervallo di confidenza al 98% di $\pm 2.082s$). Al contrario guardando alla Figura 6.14 possiamo osservare come i tempi di risposta del `server[1]` rimangano stabili durante tutta l'esecuzione (pur essendo amplificati dalla presenza del delayer) mantenendo un valore medio di $217.6ms$ e un intervallo di confidenza al 98% di $\pm 4.409ms$.

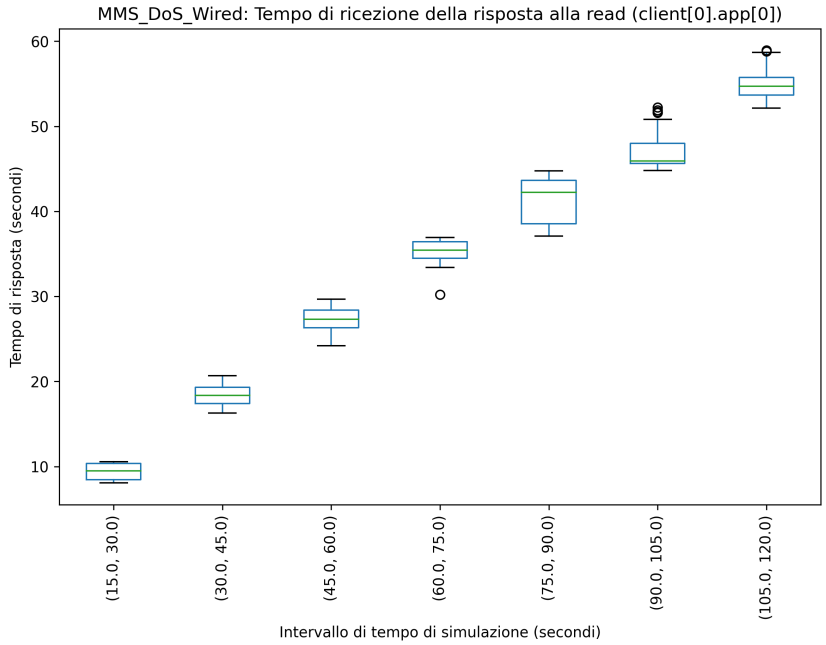


Figura 6.13: Tempo di risposta alle richieste di lettura di misura con il `server[0]`

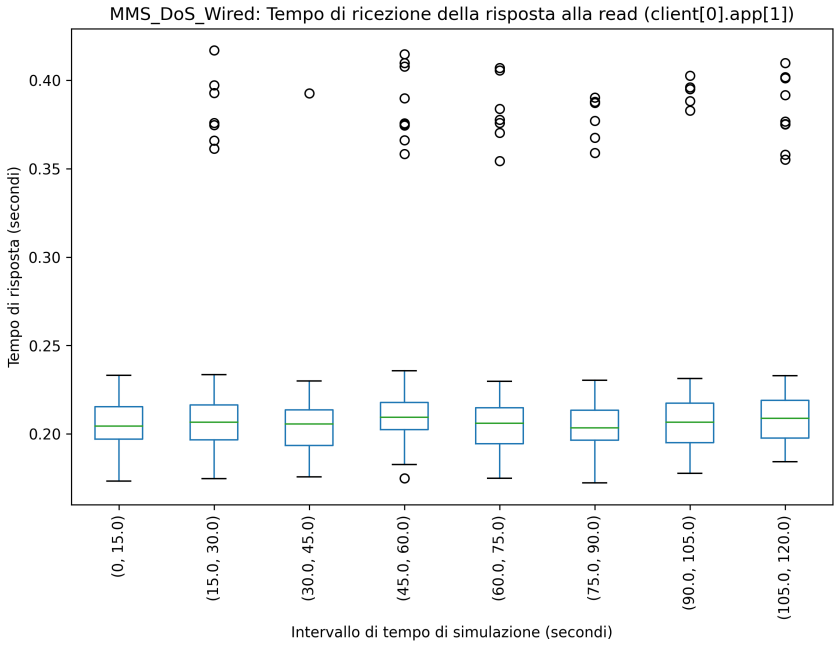


Figura 6.14: Tempo di risposta alle richieste di lettura di misura con il `server[1]`

6.3 Tempo di esecuzione

L'ultima analisi che abbiamo voluto svolgere è quella riguardante gli aspetti prestazionali delle diverse simulazioni. Le specifiche hardware e software della macchina su cui sono state eseguite le prove sono le seguenti:

- **OS**: Ubuntu 20.04.5 LTS x86_64

- **CPU**: Intel i7-3630QM (4 core e 8 thread) @ 3.4GHz
- **RAM**: 2 x 4GiB SODIMM DDR3 Synchronous 1600 MHz (0.6 ns)
- **SSD**: Kingston SSD A400 (480GB)

Prima di procedere con l'esposizione delle configurazioni su cui ci siamo soffermati in questa fase è necessario comprendere quali sono i parametri che ci hanno permesso di valutare i diversi aspetti relativi alle prestazioni di un determinato modello.

- ***runtime***: È il tempo impiegato da una run per essere eseguita.
- ***ev/sec*** (performance): Indica quanti eventi sono processati in un secondo di tempo reale ed è sia influenzato dall'hardware della macchina (avere una CPU più veloce corrisponde a poter gestire più eventi al secondo) che dalla complessità di elaborazione dei singoli eventi (costo computazionale).
- ***simsec/sec*** (velocità relativa): Specifica quanto velocemente la simulazione progredisce in relazione al tempo reale e il suo valore dipende pressoché da tutti gli aspetti della simulazione (hardware della macchina, dimensione del modello, complessità degli eventi).
- ***ev/simsec*** (densità di eventi): Rappresenta quanti eventi sono presenti per ogni secondo simulato e dipende esclusivamente dal modello e dalle sue dimensioni.

Tutte le statistiche sopra indicate verranno espresse in media e deviazione standard calcolate su 50 run d'esecuzione.

Ci siamo concentrati in particolare sulle simulazioni più gravose da eseguire per ciascuno dei due tipi di attacchi studiati e per le diverse tecnologie di comunicazione introdotte. Nelle Tabelle 6.7, 6.8 e 6.9 sono contenute le variabili sopra elencate rispettivamente per le configurazioni MMS_MITM_Wired_Cli_Atk_Delayed, MMS_MITM_Wireless_Cli_Atk_Delayed e MMS_MITM_5G_Cli_Atk_Delayed.

Variabile	***AVG***	***STD***
runtime (s)	0.150	0.029
ev/sec	83636.056	13444.471
simsec/sec	824.206	133.524
ev/simsec	101.539	3.120

Tabella 6.7: Prestazioni della configurazione MMS_MITM_Wired_Cli_Atk_Delayed

Ciò che emerge da queste prove relative all'attacco MITM è che il tempo d'esecuzione medio delle run e il numero di eventi presenti per ogni secondo simulato aumentano passando da rete cablata (0.150s e 101.539) a rete wireless (1.003s e 432.393) e a rete 5G (7.032s e 10838.128). Questa tendenza è sintomo del fatto che la simulazione di eventi relativi a protocolli di comunicazione Wi-Fi e 5G sia ben più articolata (come costo computazionale e numero di eventi) di quella relativa alle connessioni Ethernet. Questo fenomeno è anche evidenziato dalla decrescita dei

Variabile	***AVG***	***STD***
runtime (s)	1.003	0.167
ev/sec	53211.332	8885.544
simsec/sec	123.041	20.298
ev/simsec	432.393	5.821

Tabella 6.8: Prestazioni della configurazione MMS_MITM_Wireless_Cli_Atk_Delayed

Variabile	***AVG***	***STD***
runtime (s)	7.032	0.570
ev/sec	187185.620	19163.454
simsec/sec	17.272	1.776
ev/simsec	10838.128	25.801

Tabella 6.9: Prestazioni della configurazione MMS_MITM_5G_Cli_Atk_Delayed

secondi simulati per ogni secondo reale, percorrendo la stessa sequenza di configurazioni (824.206 per rete cablata, 123.041 per rete parzialmente Wi-Fi e 17.272 con connessioni 5G).

Variabile	***AVG***	***STD***
runtime (s)	4.652	0.508
ev/sec	92167.750	10800.622
simsec/sec	28.175	3.309
ev/simsec	3271.405	14.299

Tabella 6.10: Prestazioni della configurazione MMS_DoS_Wired con 10 client malevoli

Guardando invece alle Tabelle 6.10, 6.11 e 6.12, riguardo alle configurazioni per l'attacco di DDoS, possiamo notare un generale incremento dei tempi d'esecuzione causato per la maggior parte dall'introduzione dei 10 client malevoli (vedremo meglio in seguito qual è il loro effetto con dei test specifici). Un altro fattore che contribuisce all'incremento del tempo d'esecuzione è anche in questo caso portato dall'adozione di reti Wi-Fi e 5G, ma con un contributo diverso rispetto alle configurazioni MITM. Osservando infatti l'attacco DDoS nella configurazione MMS_DoS_Wireless è facile notare come il tempo d'esecuzione (79.863s) sia molto maggiore di quello registrato per la configurazione MMS_DoS_Wired_5G (22.467s) (nei modelli di MITM si verificava la situazione opposta tra rete Wi-Fi e 5G). Ciò è dato dalla presenza di *WirelessHost* sia dalla parte dei client che da quella dei server nella rete con i due *AccessPoint* Wi-Fi. Questo provoca un elevato carico di simulazione del mezzo di trasmissione e del protocollo di comunicazione ben superiore rispetto a quello presente nella rete con connessioni 5G, dove queste si trovano soltanto lato server e il

Variabile	***AVG***	***STD***
runtime (s)	79.863	10.704
ev/sec	46097.592	10583.21
simsec/sec	1.540	0.351
ev/simsec	29345.532	4331.781

Tabella 6.11: Prestazioni della configurazione MMS_DoS_Wireless con 10 client malevoli

Variabile	***AVG***	***STD***
runtime (s)	22.467	3.809
ev/sec	95187.700	21593.384
simsec/sec	5.660	1.290
ev/simsec	16818.878	141.619

Tabella 6.12: Prestazioni della configurazione MMS_DoS_Wired_5G con 10 client malevoli

numero di host che le sfruttano è inferiore.

Variabile	***AVG***	***STD***
runtime (s)	12.249	1.243
ev/sec	134009.576	25214.109
simsec/sec	9.624	1.809
ev/simsec	13922.246	55.612

Tabella 6.13: Prestazioni della configurazione MMS_DoS_Wired_5G con 5 client malevoli

Un aspetto già accennato su cui abbiamo voluto approfondire è quello riguardante il numero di client malevoli presenti nella rete. Quello che emerge dai test visibili nelle Tabelle 6.13 e 6.14 e relativi alla configurazione MMS_DoS_Wired_5G, rispettivamente con 5 e con 20 client malevoli, è che il tempo d'esecuzione tende a crescere linearmente con l'aumentare dei moduli compromessi. Quindi se il numero di client malevoli raddoppia (da 5 a 10) allo stesso modo segue il tempo di simulazione (passa da $12.249s$ a $22.467s$) e di conseguenza i secondi simulati per ogni secondo reale si dimezzano (da $9.624s$ a $5.660s$). Il valore relativo alla densità di eventi (ev/simsec) invece aumenta al crescere dei client malevoli siccome è sempre in relazione alla complessità del modello (per esempio se i client compromessi passano da 5 a 10 allora gli eventi gestiti per ogni secondo simulato passano da 13922.246 a 16818.878).

Variabile	***AVG***	***STD***
runtime (s)	41.288	2.473
ev/sec	52983.782	20876.497
simsec/sec	2.359	0.922
ev/simsec	22416.036	401.014

Tabella 6.14: Prestazioni della configurazione MMS_DoS_Wired_5G con 20 client malevoli

Capitolo 7

Conclusioni e sviluppi futuri

Questa tesi considera il problema di modellare ad un livello di astrazione adeguato le comunicazioni che avvengono tra i sistemi DER e i centri di controllo che ne regolano il funzionamento e alcuni (possibili) tipi di cyberattacchi mirati a destabilizzare il sistema di distribuzione dell'energia elettrica che integrano i DER. Il lavoro svolto ha permesso di sperimentare alcuni ambienti di simulazione orientati alla rappresentazione di sistemi distribuiti e dotati di moduli in grado di modellare piuttosto dettagliatamente i protocolli dello stack TCP/IP su reti cablate (Ethernet) o senza fili (Wi-Fi) oltre che a reti mobili come il 5G. Uno degli obiettivi del lavoro svolto era di creare una serie di modelli parametrici e facilmente configurabili capaci di rappresentare scenari realistici definiti in collaborazione con RSE. Tali modelli dovranno affiancare un testbed realizzato da RSE per poter studiare vari tipi di attacchi conosciuti e sviluppare dei sistemi evoluti d'Intrusion Detection. Più in dettaglio è stato individuato un sottoinsieme sufficientemente significativo delle funzionalità offerte dal protocollo MMS e sono stati costruiti dei modelli di simulazione basati su *OMNeT++*, integrati con i moduli dello stack TCP/IP implementati dalla libreria *INET*. In seguito sono stati modellati attacchi di tipo Distributed Denial Of Service e di Man In The Middle integrando nuovi componenti nei modelli di partenza, definendo come questi avrebbero dovuto comunicare tra loro. Nell'ultima fase abbiamo sperimentato diverse tecnologie di comunicazione come quella Wi-Fi (già offerta all'interno di *INET*) e quella 5G (sviluppata nella libreria *Simu5G*). Nella parte conclusiva della tesi ci siamo focalizzati sulla raccolta di dati che permettessero di studiare l'impatto sul comportamento del sistema e sulle misure in grado di evidenziare gli effetti dei diversi attacchi.
Un possibile sviluppo futuro di questo lavoro prevede d'integrare un meccanismo di generazione semiautomatica dei modelli di simulazione a partire dai componenti di base costruiti o da loro ulteriori estensioni: l'idea è di fornire in input la topologia delle reti da un lato (a partire da formalismi grafici o testuali) e il comportamento dell'attaccante dall'altro (sulla base di opportuni grafi d'attacco), in modo da poterne studiare le diverse combinazioni.
Un'altra questione ancora aperta riguarda come potrebbe essere sfruttato questo progetto nello sviluppo di un sistema d'*Intrusion Detection* efficace. Come dettagliato nelle descrizioni dei modelli di `client`, `server` e attaccante (specialmente in relazione all'attacco MITM) possiamo notare in ciascuno di essi la presenza di un **logger** in grado di registrare su file delle tracce di pacchetti transitati e la loro evoluzione nel tempo. Nelle nostre simulazioni abbiamo un'osservabilità totale di

ciò che accade, quindi grazie a questi file di log possiamo sapere in ogni istante cos'è accaduto a un determinato messaggio MMS anche in relazione allo stato dei vari moduli nella rete (operatività del server, forza di attacco dell'`evilClient` ecc...). Attraverso un'opportuna fase di elaborazione successiva questi log grezzi possono essere trasformati in dataset di training per modelli d'ID. Un esempio pratico in fase di studio riguarda l'apprendimento delle tabelle di probabilità condizionata (CPT) di Reti Bayesiane Dinamiche (DBN) che modellano i diversi tipi d'attacco partendo dai log. Date le tracce restituite dal simulatore è stato realizzato un prototipo che le converte in serie temporali di valori di variabili associate ai nodi della DBN per ogni time-slice. È poi compito dell'algoritmo di *Expectation Maximization* di andare a stimare i valori nelle CPT che meglio descrivono i dati osservati. Come spiegato in sezione 2.1 questo tipo di modelli può essere utilizzato nella rilevazione delle intrusioni andando a determinare la probabilità che l'attaccante intraprenda determinate azioni di attacco sulla base delle osservazioni fornite e della situazione della rete negli istanti di tempo precedenti.

Bibliografia e Sitografia

[1] Davide Cerotti et al. ≪A Modular Infrastructure for the Validation of Cyberattack Detection Systems≫. In:
Power Systems Cybersecurity: Methods, Concepts, and Best Practices.
A cura di Hassan Haes Alhelou, Nikos Hatziargyriou e Zhao Yang Dong.
Cham: Springer International Publishing, 2023, pp. 311–336.
ISBN: 978-3-031-20360-2. DOI: 10.1007/978-3-031-20360-2_13.
URL: https://doi.org/10.1007/978-3-031-20360-2_13.

[2] The MITRE Corporation.
Adversarial Tactics, Techniques and Common Knowledge (ATT&CK). 2015.
URL: https://attack.mitre.org/.

[3] The MITRE Corporation. *ATT&CK for containers*. 2021. URL: https://collaborate.mitre.org/matrices/enterprise/containers/.

[4] The MITRE Corporation. *ATT&CK for industrial control systems*. 2020.
URL: https://collaborate.mitre.org/attackics/index.php/Main_Page.

[5] Vittorio Farina.
≪Sviluppo in Omnet++ di modelli di attacco a MQTT e MMS≫.
Tesi di Laurea. Università del Piemonte Orientale, 2022.

[6] *INET User's Guide*.
URL: https://inet.omnetpp.org/docs/users-guide/.
(ultima visita: 20.03.2023).

[7] Ansam Khraisat et al.
≪Survey of intrusion detection systems: techniques, datasets and challenges≫.
In: 20 (2019), pp. 1–12.

[8] Matsumoto e Takuji Nishimura. *Mersenne Twister*. 1997. URL:
http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/emt.html.

[9] Giovanni Nardini et al.
≪Simu5G: A System-level Simulator for 5G Networks≫.
In: *SIMULTECH* (2020), pp. 1–13.

[10] *OMNeT++ Simulation Manual*.
URL: https://doc.omnetpp.org/omnetpp/manual/.
(ultima visita: 15.04.2023).

[11] Mauro José Pappaterra e Francesco Flammini.
≪Bayesian Networks for Online Cybersecurity Threat Detection≫. In:
Machine Intelligence and Big Data Analytics for Cybersecurity Applications.
A cura di Yassine Maleh et al.
Cham: Springer International Publishing, 2021, pp. 129–159.
ISBN: 978-3-030-57024-8. DOI: 10.1007/978-3-030-57024-8_6.
URL: https://doi.org/10.1007/978-3-030-57024-8_6.

[12] Inc. Systems Integration Specialists Company. ≪Overview and Introduction to the Manufacturing Message Specification (MMS)≫. In: (1995), pp. 1–18.

[13] Mauro G. Todeschini, Giovanna Dondossola e Roberta Terruggia.
≪Impact Evaluation Of IEC 62351 Cyber Security On IEC 61850 Communications Performance≫.
In: *CIRED 2019 Conference* 1917 (2019), pp. 1–5.

[14] R. Vinayakumar et al.
≪Deep Learning Approach for Intelligent Intrusion Detection System≫.
In: *IEEE* (2019), pp. 1–26.