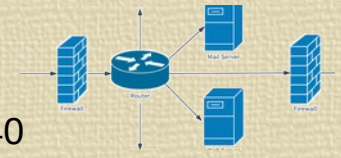
PaketTracer: no, grazie...

Come fare esercitazione di reti in un mondo libero

Ing. Stefano Salvi – stefano@salvi.mn.it

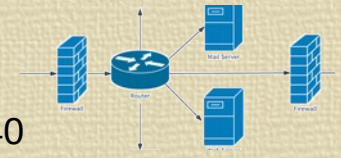


LUGMan

LINUX DAY ONLINE
24/25 OTTOBRE
www.linuxday.it

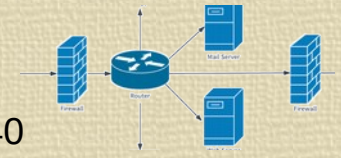
Il problema

- Insegno Sistemi e reti in un Istituto Superiore
- Devo fare esercitazioni di configurazione macchine, servizi e reti
- Far configurare le macchine del laboratorio ai ragazzi non è consigliabile
- Le macchine del laboratorio non hanno tutti i servizi installati e una volta fatto (magari sbagliato) l'esercizio le macchine potrebbero essere da risistemare
- Per le reti la soluzione classica è l'uso di Cisco PacketTracer, che non è open source e gira sotto Linux solo con Wine



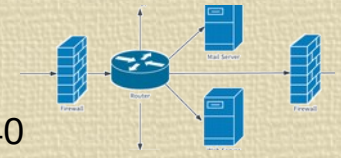
La soluzione immaginata

- L’optimum sarebbe creare una rete virtuale di macchine virtuali, che possono essere configurate, collegate e testate
- Per creare una situazione di rete da configurare/verificare occorrerebbero anche 5 macchine virtuali o più con configurazioni diverse oltre a quattro o cinque reti virtuali
- Con soluzioni di virtualizzazione classica le risorse di sistema richieste sono proibitive



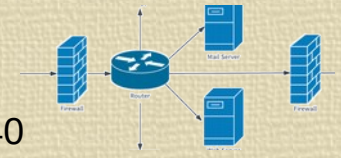
Usermode Linux

- Una soluzione interessante è usermode linux (d'ora in poi UML) che è un "flavour" del kernel che, invece di appoggiarsi all'hardware tramite le driver, si appoggia al sistema operativo ospite.
- Questo kernel viene lanciato come una normale applicazione e funziona in spazio utente
- Di norma il file system viene immagazzinato in un file, come nelle virtualizzazioni classiche, ma ci sono anche altre soluzioni.



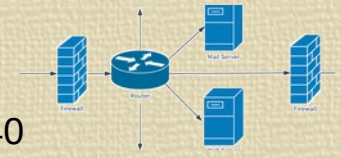
I dischi

- Di norma, come dicevo, UML utilizza come dispositivi a blocchi (dischi) dei file regolari.
- UML si aspetterà che questi dispositivi contengano un file sytem tra quelli configurati nel kernel (quasi tutti i file system comuni di Linux)
- Durante la fase di boot uno di questi viene montato come root dal kernel che carica da questo il sistema, come in un normale boot.
- Per gestire questi file dal sistema host (ovviamente quando le macchine virtuali sono spente) si possono montare con il dispositivo *loop* (`-o loop`). Naturalmente occorre avere i privilegi di root per farlo.



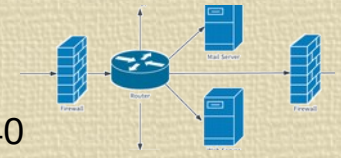
Readonly e Read-write

- Se devo lanciare contemporaneamente più macchine, ognuna deve avere il suo disco, in quanto ognuna lo deve poter scrivere.
- Per ridurre il peso sull'host è anche possibile creare un disco *copy on write*. In questo caso il file originale è aperto in sola lettura ma ogni istanza di UML crea un file con lo stesso nome del file e l'estensione **.cow** nel quale scriverà le sole modifiche che la macchina virtuale farà.
- In alternativa posso montare il disco in modalità read only, ma il sistema deve essere configurato per avere il dispositivo root in sola lettura.
- Un inconveniente delle vecchie versioni di UML (del file system ext4) era (credo lo abbiano risolto) che anche se associo un dispositivo in read-only, UML cerca il file **.cow** e se non lo trova cerca di crearlo, cosa che crea problemi se il disco originale è in un'area dove l'utente non ha diritti di scrittura. Ora non lo fa più, per lo meno se il disco è formattato con *squashfs*



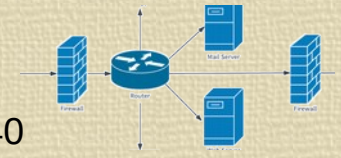
Directory dell'host

- Maneggiare le immagini di disco non è agevole mentre per correggere gli esercizi servirebbe avere i file modificati direttamente disponibili nella macchina del docente.
- UML mette a disposizione un file system *hostfs* che consente di montare una directory come se fosse un dispositivo.
- L'inconveniente di questo approccio è che l'accesso ai file viene fatto dall'utente e quindi tutti i file nella directory devono avere gli opportuni diritti per essere gestiti dall quell'utente, anche se da dentro la macchina opererò come root.
- Inoltre la macchina virtuale vede il proprietario reale dei file e quindi potrei avere dei problemi di accesso se l'UID dell'utente della macchina virtuale non coincide con l'UID della macchina reale.
- Alcuni demoni poi controllano che i file di configurazione siano di root e non accessibili agli utenti. I file in *hostfs* appaiono con il loro utente reale e questo controllo fallisce.
- Il problema viene risolto con *humfs*, che crea un metafile contenente i proprietari, gruppi e diritti per file e directory. Purtroppo questo file system non è di solito compilato nei kernel UML delle distribuzioni



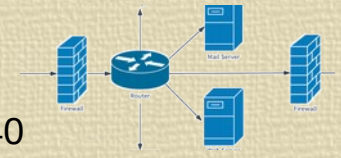
La configurazione delle macchine

- Per prima cosa, creiamo un disco contenete, oltre al sistema operativo, tutte le utility ed i servizi che ci interesserà utilizzare e configurare.
- Scegliamo poi di creare una root in sola lettura, per non doverci preoccupare dei file **.cow** per le singole macchine. Scegliamo *squashfs*, che oltre ad essere in sola lettura ci consente di risparmiare spazio su disco.
- Non è disponibile *unionfs* (o qualcosa di simile) nel kernel, quindi particolare cura andrà messa nel creare dei ramdisk per ogni parte scrivibile nel sistema. In questo senso le nuove distribuzioni ci vengono in aiuto in quanto tutte le directory temporanee sono già installate in ramdisk, lasciando poco da configurare. Rimane la `var`, di cui parleremo in seguito



Le *etc* personalizzate

- Abbiamo detto che lo scopo del sistema è di consentire agli studenti di esercitarsi a configurare le macchine, ma le configurazioni sono nella directory `etc`.
- Possiamo montare la `etc` ma dobbiamo ricordare che fino al momento in cui viene eseguito lo scrpt di mount `mountall.sh`, la etc è quella dello squashfs.
- Se però la montiamo da un file sytem scrivibile abbiamo due vantaggi:
  1) Possiamo consentire agli studenti di creare/modificare le configurazioni del sistema
  2) Possiamo creare delle configurazioni precostituite per creare un esercizio, ad esempio possiamo creare un server configurato con DHCP, DNS e HTTP per fare esercizi sulle configurazioni

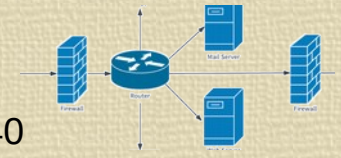


Etc di base e patch

- Per ridurre l'impatto sul disco (ed il salvataggio dell'esercizio svolto) ho deciso di usare un hostfs per la `etc`.
- La directory che verrà montata come `etc`, ovviamente una per macchina, viene inizializzata con un file *tar.bz2*.
- Nel caso la macchina debba avere una confiugrazione personalizzata, al contenuto decompresso viene applicata una patch.
- Questo porta due vantaggi:

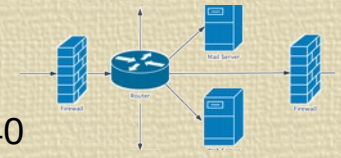
  1) Riduce pesantemente la dimensione dei dati che costituiscono un esercizio

  2)Consente di aggiornare il sistema e la etc di base senza interferre con gli esercizi già creati.



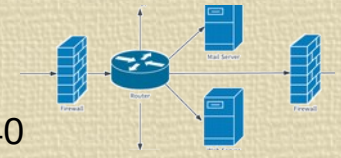
L'inconveniente delle `etc` come hostfs

- Usare hostfs per la directory `etc` è estremamente comodo sia per la leggibilità che per la possibilità di aggiornamento.
- Presenta però l'inconveniente che alcuni programmi (ad esempio ftp o ssh) verificano il proprietario ed i diritti di alcuni file di configurazione.
- Per risolvere il problema ho dovuto modificare i file di avvio di questi servizi, creando i file di configurazione che devono essere di un certo utente in `/tmp` con proprietario e diritti corretti e sostituendo i file in etc con dei link simbolici a questi.
- In alcuni casi (ad esempio ssh) ho dovuto modificare il file di configurazione per eliminare alcuni controlli di sicurezza.



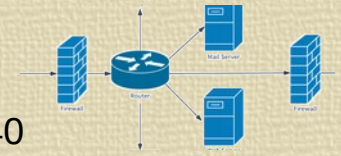
La directory `var`

- Una directory dove il sistema scrive e dove occorre scrivere per utilizzare alcuni servizi è la directory `var`.
- In essa troviamo i log che ci servono per individuare eventuali errori
- Ovviamente essa va tolta dall'immagine del sistema operativo, che è in sola lettura.
- Per non appesantire troppo il montaggio del sistema ho deciso di non creare un ulteriore volume per la `var`. Ho invece copiato la var in una sottodirectory di `etc` (`/etc/var.moved`) e sostituito la directory `var` dell'immagine con un link simbolico a questa sottodirectory.



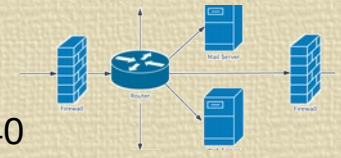
Lo scenario: macchine e reti

- Se voglio creare un esercizio, devo creare una serie di macchine e reti.
- Ad esempio, se volessi creare un esercizio realistico sulla NAT, mi servirebbero:
  - Due router che facciano NAT
  - Un router che li colleghi, per evitare connessioni dirette
  - Due client nelle due reti nattate
- Dovrei poi creare quattro reti
  - Le due reti nattate con indirizzi privati
  - Le due reti con indirizzi reali che collegano i due router che fanno nat



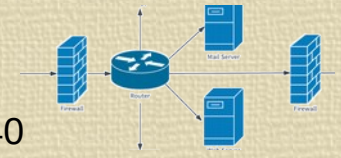
Le configurazioni delle macchine

- Se lo scopo dell'esercizio è di sperimentare i cambiamenti di indirizzo e non la configurazione, le macchine dovrebbero essere tutte preconfigurate.
- In teoria dovremmo produrre etc diversi per ogni macchina, ma in realtà:
  - I due client delle reti nattate richiedono solo al configurazione dell'unica interfaccia di rete (indirizzo/maschera/gateway o dhcp)
  - Il router centrale prevede solo al configurazione delle due interfacce (indirizzo/maschera/gateway) e l'abilitazione del forwarding
  - Solo i due router che fanno NAT hanno bisogno di una configurazione particolare, in quanto occorre configurare le regole di firewall oltre alle interfacce.
- Per evitare di creare una configurazione per ognuna delle macchine coinvolte sarebbe interessante poter passare un comando al kernel (o meglio al sistema) che imposti le configurazioni standard alla creazione della macchina.



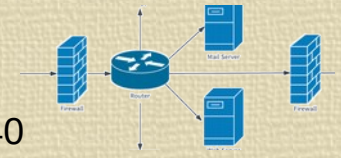
Lo script di avvio

- Modificheremo `/etc/init.d/rc.local` per analizzare ed interpretare la riga di comando del kernel, reperibile in `/proc/cmdline`.
- Cercherà nella riga di comando dei parametri il cui nome sia:
  - **eth<n>**. Questi parametri contengono di norma la configurazione dell'interfaccia di UML, ma vi ho aggiunto alcune ozioni:
    - dhcp per configurare l'interfaccia con dhcp
    - <ip>/<prefix_lenght> per configurare l'interfaccia con indirizzo fisso.
  - **nameserver** che indicherà il nome del server DNS da aggiungere in resolv.conf, per la configurazione manuale
  - **route** che conterrà una delle rotte espressa come <ip>/<prefix_lenght>;<gateway>
  - **forward** che imposterà il forwarding attivo
  - **hostname** che imposterà il nome della macchina (cui il testo dell'esercizio farà riferimento) che quindi non sarà configurabile.



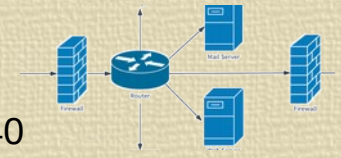
Iptables ed i moduli

- Volendo anche configurare il firewall, ci serve caricare alcuni moduli di iptables a runtime.
- Per fare questo occorre che la directory /lib/modules/<versione del kernel> sia disponibile.
- Quando creo l'immagine del sistema virtuale, non so quale versione del kernel sarà installata nel sistema host, quindi non posso precaricare la directory dei moduli
- I moduli di UML aggiornati sono comunque installati nel sistema host in /usr/lib/uml/modules/
- La logica vorrebbe che noi montassimo questa directory come hostfs
- Purtroppo il filesystem hostfs può montare solo sottodirectory di directory specifica per macchina virtuale
- Questa sarà la directory privata della macchina virtuale, contenente la etc e le home degli utenti root e user
- Salvo che noi non vogliamo pubblicare / per la nostra macchina virtuale, la directory dei moduli non sarà contenuta in questa directory



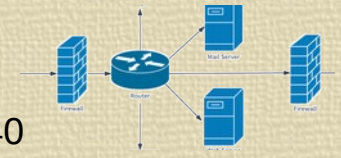
Usermount e bind

- Il filesystem hostfs non seguie i link simbolici, ma li esporta come sono nella macchina virtruale, quindi un symlink alla directory dei moduli nella directory dell'host non consentirebbe di trovare i moduli nella macchina virtuale
- Una soluzione alternativa valida è quella di montare in bind la directory dei moduli nella directory della macchina, ma l'operazione di mount è riservata a root.
- Per fortuna esiste un pacchetto usermount che consente ad un utente regolare di eseguire una mount, pur con certe limitazioni.
- Un inconveniente di questo metodo è che a volte le directory montate non vengono liberate immediatamente alla chiusura della macchina virtuale e quindi a volte lo smontaggio delle directory montate in bind non ha successo, ma comunque al reboot verranno eliminate.



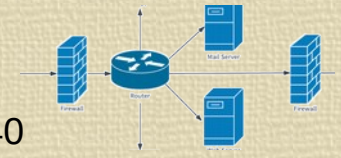
Il layout definitivo dei dischi

- hostfs punta alla directory della macchina /tmp/network.<xxx>/machines/<mx>/
- **/usr/share/umlnetsym/netimage_amd64-root_fs.squash** montato su **/**
- **hostfs/etc** montato su **/etc**
- **hostfs/modules** montato su **/usr/lib/modules**
- **hostfs/root** montato su **/root**
- **hostfs/user** montato su **/home/user**



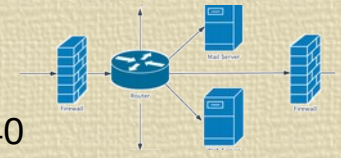
Probelmi all'avvio

- Gli script di avvio utilizzano i dati di /proc prima che il file system virtuale sia montato
- /dev/random di base non funziona in UML. Occorre usare urandom al suo posto
- Nella nostra implementazione, utilizzando hostfs che non rispetta i proprietario e gruppo della macchina virtuale, bisogna assicurarsi che tutti i file in /var siano scrivibili
- Tutti questi problemi devono essere risolti prima che partano gli script di avvio in /etc/intit.d
- Per fortuna init non li lancia direttamente ma utilizza lo script /lib/init/rcS che noi potremo modificare per risolvere questi problemi prima dell'inizializzazione.



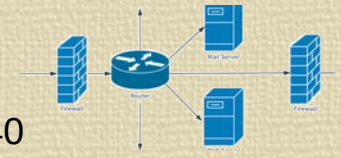
Gli xterm

- Se UML viene lanciato in un ambiente grafico, i terminali della macchina virtuale possono essere visualizzati dentro a dei terminali virtuali.
- Per associare alla console o ad un altra linea seriale una finestra di terminale basta indicarlo sulla riga di comando.
- Il terminale utilizzato è **xterm** (il terminale base di x-windows) e non quello del Desktop Environment utilizzato.
- È comunque possibile indicare un altro emulatore di terminale con il parametro `xterm=`, che però è alquanto rigido.
- Per rendere le cose più elastiche, invece che lanciare un terminale direttamente, con il parametro `xterm=` indico uno script che imposta il titolo in maniera automatica e mi consente di scegliere e configurare a mio piacimento il terminale virtuale.



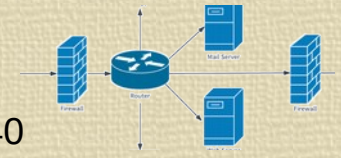
Più utenti e più temrinali

- Parecchi esercizi prevedono l'uso di un utente regolare, dotato di password, oltre a root (ad esempio ssh con password non va con root nelle configurazioni standard)
- Inoltre alcune operazioni richiedono di operare con più di un terminale, ad esempio salvare una traccia con tcpdump oppure usare il port forwarding di SSH diventa parecchio brigoso se si ha a disposizione un solo terminale.
- UML consente di avere quanti terminali si vuole, ma ognuno occuperà una diversa finestra, rendendo molto difficile capire su che macchina virtuale si sta operando.



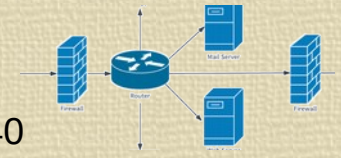
Creare due tab

- La soluzione semplice e ovvia è quella di creare un solo terminale con più tab, cosa permessa da tutti i moderno desktop environment.
- È sempre anche possibile creare più tab in un terminale tramite riga di comando dal terminale stesso.
- È spesso possibile da riga di comando lanciare un comando in una diversa tab (con Cinnamon non è possibile – non ho valutato KDE).
- Il problema è come mandare i vari terminali di una singola macchina virtuale in tab diversi dello stesso xterm.



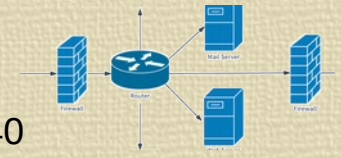
Il meccandismo di avvio degli xterm

- Per trovare la soluzione occorre per prima cosa capire il meccanismo con cui UML crea gli xterm.
- Quando UML inizializza un terminale lancia il relativo xterm passandogli, come comando da eseguire, un helper che ha come parametro un riferimento ad un socket attraverso cui comunicherà con il la linea seriale della macchina virtuale.
- Il socket è deciso al momento dell'avvio del terminale, quindi non è possibile prevederlo in anticipo e lanciare l'helper in attesa che UML inizializzi il terminale e il socket.



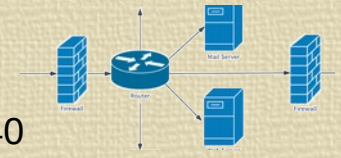
L’inizializzazione dei tab

- Non è possibile collegarsi ad un terminale aperto ed aggiungere una tab
- Il terminale deve essere creato fin dall’inizio con due tab, ma al momento della creazione non è disponibile il socket a cui connettere l’helper.
- La soluzione trovata è:
  - Creare per ogni macchina un terminale con due tab
  - Creare una named pipe (fifo) per ogni tab
  - Inizializzare le tab con uno script che ascolti la named pipe in attesa del riferimento al socket e, all’arrivo, lanci l’helper con il giusto parametro
  - Impostare gli xterm in modo che invece di avviare un terminale, lancino uno script che scriva il riferimento al socket sulla named pipe
- Dato che lo script dell’xterm non può essere personalizzato, la named pipe viene chiusa immediatamente dopo l’utilizzo e lo script scrive sulla prima delle due named pipe che trova ancora aperta.



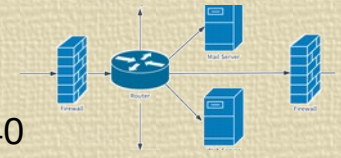
La descrizione della rete

- Ora che abbiamo la struttura delle macchine, occorre trovare un modo per creare un esercizio con queste macchine.
- Occorre creare un file che descriva le macchine e gli switch da costruire e ne indichi le caratteristiche
- Dovremo poi avere i file patch per le etc personalizzate ed un file PDF con le istruzioni per lo studente
- Utilizzeremo un file tar.gz rinominato in .umlnz per contenere il file di descrizione con tutti i file accessori appena descritti



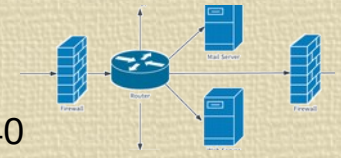
Il linguaggio di descrizione della rete

- Sostanzialmente la descrizione della rete e dell'esercizio viene realizzata tramite un file con estensione **.umlnetsym** che contiene poche righe di configurazione:
  - **switch**: descrive uno switch virtuale che collegherà alcune delle nostre macchine
  - **macchina**: descrive una delle macchine del nostro esercizio, router, server o workstation
  - **docfile**: indica il file PDF con il testo dell'esercizio
- Ognuna di queste ha i suoi parametri che sono una lista separata dal carattere '**:**' di coppie *nome=valore* dove il valore può anche essere strutturato.



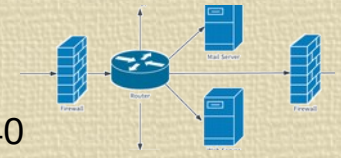
Le configurazioni di **switch** e **docfile**

- Queste configurazioni sono abbastanza semplici ed hanno pochi parametri
  - **switch**:
    - **nome** (obbligatorio) indica il nome con cui lo switch verrà indicato per connetterlo all'interfaccia di una macchina
    - **descr** indica la descrizione dello switch che comparirà nello schema della rete
    - **indirizzo** indica l'indirizzo della subnet gestita dallo switch e serve solo come documentazione – compare nello schema.
  - **Esempio**: switch nome=Insecure:descr=Rete Insicura:indirizzo=80.0.0.0/8
  - **docfile**:
    - (obbligatorio) Il nome del file PDF con la descrizione dell'esercizio
    - **Esempio**: docfile=esercizio_dmz.pdf



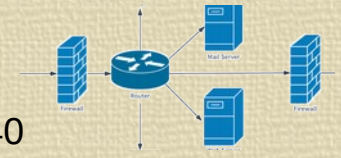
L'istruzione **macchina**

- Questa configurazione è la più complessa e descrive le macchine che costituiscono l'esercizio e le loro connessioni, ha molti parametri:
  - **macchina**:
    - **nome** (obbligatorio) il nome dell'host, che comparirà anche nel prompt
    - **descr** la descrizione che comparirà nella barra del titolo del terminale e nello schema della rete
    - **rete** contiene un elenco di interfacce di rete con le loro caratteristiche, separate da ','. Ogni interfaccia avrà questa struttura: <*nome_interfaccia*>**->**<*switch*>**;**<*configurazione*> dove:
      - ***<nome_interfaccia>*** (obbligatorio) è il nome dell'interfaccia, che sarà chiamata eth<numero> (eth0, eth1, eth2) nella nostra macchina
      - ***<switch>*** (obbligatorio) è il nome dello switch cui collegarla, come indicato nell'istruzione switch
      - ***<configurazione>*** (opzionale) la configurazione dell'interfaccia. Se manca, l'interfaccia dovrà essere configurata a mano dallo studente oppure tramite il file /etc/network/interfaces (che dovrà essere messo in una *etc* personalizzata). Altrimenti può avere queste due forme:
        - **<ip>/<lung.prefisso>** l'IP fisso da assegnare alla rete, con la dimensione del prefisso (notazione CIDR)
        - **dhcp** indica che l'interfaccia sarà configurata da un server dhcp (che dovrò aver configurato su di un altro host della rete).
    - **forward** è un flag: se presente abilita il forwarding IPv4 tra le interfacce
    - **mem** indica la quantità di memoria da associare alla macchina se i 120M di default non vanno bene
    - **Esempio:** macchina nome=FW:descr=Firewall:rete=eth0->Insecure;80.90.112.96/8,eth1->Secure;192.168.1.1/24,eth2->DMZ;192.168.2.1/24:forward



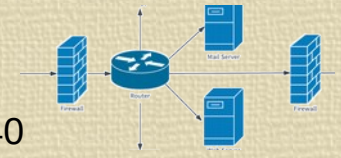
Lo script che crea il file dell'esercizio

- La descrizione dell'esercizio richiama file esterni e ne richiede altri. Sarà opportuno creare una directory per l'esercizio che contenga tutti i suoi file.
- Fatto questo potremo processare la descrizione della rete **<esercizio>.umlnetsym** con il **compilatore** di esercizi, che cercherà tutti i file necessari e li raccoglierà nel file **<esercizio>.umlnz** che sarà in realtà un archivio **tar.bz2**.
- Lo script:
  - Interpreta la descrizione individuando gli eventuali errori come parametri non corretti, switch mancanti, file di descrizione dell'esercizio mancante
  - Per ogni macchina definita cerca un file **<macchina>.patch** con il nome della macchina, che personalizzerà la sua etc
  - Se questo file manca, cerca un file **<macchina>.tar.bz2**, lo decomprime in una directory temporanea a fianco del file etc di base e crea da questi il file **<macchina>.patch**.
  - Finita questa scansione crea il file dell'esercizio **<esercizio>.umlnz** contente
    - Il file **<esercizio>.umlnetsim**
    - Il file **PDF** di descrizione dell'esercizio indicato nel file precedente
    - Tutti i file **<macchina>.patch** trovati o creati, per le macchine indicate nell'esercizio.



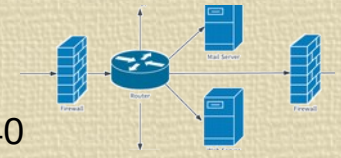
Lo schema automatico della rete

- Dovendo fare una esercitazione sulle reti è quasi sempre necessario mostrare lo schema della rete.
- Per evitare di dover disegnare lo schema in ogni esercizio, allegandolo alla documentazione PDF, ho individuato **graphwiz**, che è un “software di vilualizzazione di grafi”.
- Presenta più moduli, ciascuno che utilizza una diversa disposizione per rappresentare il grafo, che nel nostro caso è la rete.
- Il grafo deve venire descritto con un linguaggio che ne indica i nodi, le loro caratteristiche (forma, colore, testi) e le connessioni, sempre con le loro caratteristiche.
- A partire da questa descrizione, il modulo di **grapwiz** scelto (nello specifico **twopi** che crea una disposizione radiale) dispone nodi e connessioni.
- **twopi** salva il disegno del grafo in una immagine *png*



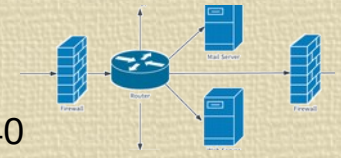
La directory di lavoro

- Umlnetsym crea una directory in /tmp con tutto il necessario per far funzionare le macchine virtuali. Conterrà:
  - Una directory testo nella quale sarà scompattato l'esercizio
  - Il file *.dot* per **twopi** con la descrizione della rete, prodotto durante il parsing della descrizione dell'esercizio
  - Il file **<esercizio>.png** prodotto da **twopi**
  - Un file d'appoggio **machines.names** con l'elenco dei nomi e delle descrizioni delle macchine utile per la comunicazione tra le parti del programma
  - Un file di appoggio **switches.pid** con i *PID* dei processi di tutti gli switch, per poterli arrestare alla fine
  - Un file per ogni macchina chiamato **<macchina>.pid** con il *PID* del processo principale della macchina, per poterla fermare tramite una **kill**, in caso di bisogno
  - I socket di ognuno degli switch virtuali, creati da **UML**
  - Una directory **machines** contenete una sottodirectory per ogni macchina, con il nome della macchina, contenente:
    - La directory **etc** (che verrà montata in **/etc** nella macchina virtuale) con i file scompattati dal file originale e patchati con l'eventuale file *.patch* di personalizzazione oppure dal file salvato all'ultima esecuzione dell'esercizio
    - La directory **modules** (che verrà montata in **/lib/modules**) nella quale è montata in bind la directory dei moduli di **UML**
    - Le directory **root** e **user** (montate rispettivamente in **/root** e **/home/user**) che saranno le home degli utenti *root* e *user* rispettivamente.
- Questa directory verrà cancellata quando verrà chiusa l'ultima macchina dell'esercizio.



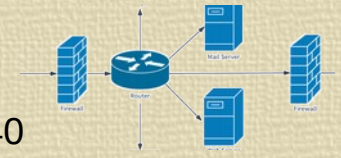
La directory dell'utente

- Nela home dell'utente umlnetsym crea una sua directory dati **~/umlnetsym** nella quale registra:
  - I file di log **umlnetsym.log** ed **umlnetsym.err** con i mssaggi e gli errori dell'ultima esecuzione dell'emulatore
  - Una directory per ogni esercizio lanciato contenente
  - Un file **<macchina>.tar.bz2** contenete la etc della rispettiva macchina, immagazzinata nel momento dello spegnimento della macchina virtuale. Vengono sovrascritti ad ogni esecuzione dell'esercizio e servono per poter continuare un esercizio non terminato
  - Un file **<esercizio>-<utente>-<data>-<ora>.tar.bz2** per ogni esecuzione dell'esercizio, salvato allo spegnimento dell'ultima macchina, contente la directory **machines** della directory temporanea vista prima. Questo file potrà essere consegnato per correggere e valutare l'esercizio ed anche ricaricato per verificare il corretto funzionamento



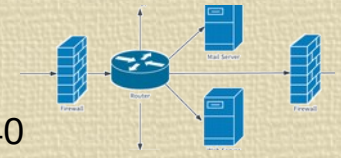
La tecnologia adottata

- Come realizzare il programma (o la suite di programmi) che realizza tutto ciò?
- Ho voluto tener conto di due problematiche:
  - Poter configurare servizi oltre che reti (gli altri simulatori disponibili sono concentrati solo sulla rete)
  - Utilizzare esclusivamente eseguibili standard disponibili tra i pacchetti delle distribuzioni
- La scelta è caduta (lo ammetto è stato più un esercizio di stile – perl o pyton sarebbero state valide alternative) sulla bash: facciamo tutto solo con script di shell.
- Comunque gli emulatori di terminale, i programmi di visualizzazione di PDF e immagini, il generatore di grafici, usermount ed i suoi moduli, patch ed UML stesso sono pacchetti terzi che vanno installati, disponibili nelle maggiori distribuzioni
- Oltre a questi ho utilizzato solo **zenity** per presentare dati all'utente o per chiedere delle scelte, oltre ovviamente ai comandi **grep**, **sed**, **[** ed **expr e** che sono gli strumento fondamentali della programmazione di shell



Lo script

- Lo script che fa funzionare l'esercizio (umlnetsym) è unico e non richiede altri file di appoggio, per semplificare l'installazione
- Questo script eseguirà i vari compiti:
  - interpretare ed avviare l'esercizio, con il testo, l'immagine della rete, le macchine virtuali ed i terminali con i due tab che saranno associati ad ognuna di esse
  - terminare le macchine virtuali in maniera forzata
  - intercettare l'avvio del terminare da parte di UML ed inviare tramite la pipe il riferimento al canale al terminale che lo deve gestire
  - Attendere nel terminale la connessione da parte di UML per avviare l'helper del terminale
- Oltre a questo avremo uno script (il compilatore) per produrre un esercizio a partire dalla descrizione, dal file con le istruzioni e dalle eventuali personalizzazioni
- Ci saranno poi piccoli script di aiuto per il manutentore per fare operazioni come generare una nuova immagine di sistema o una nuova immagine etc, utili se si deve aggiornare o modificare l'immagine di sistema comune

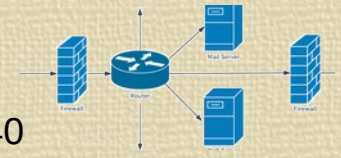


Il parsing in bash

- Un aspetto particolare è vedere come interpetare un file di descrizione tramite comandi bash
- Per prima cosa leggeremo il file riga per riga tramite un ciclo

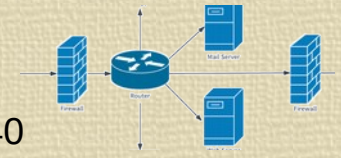
```
while read line
do
    ….
done < "$NETDEF"
```

- Ogni riga comincia con una parola chiave (anche la # di commento può essere considerata una parola chiave). Utilizzo una cascata di if ognuno che controlla la prima parola con un grep, ad esempio `if echo $line | grep -iq "^switch"` mi consente di individuare le linee che descrivono gli switch
- Una volta individuato il tipo di riga isolo i parametri con il comando `S=`echo ${line} | sed "s/^switch[\\t]*/:/i"`:` che aggiunge un terminatore `:` prima e dopo i parametri per semplificare l'interpretazione
- A questo punto potrò cercare i singoli parametri in base al loro nome con un comando come `echo $S | grep -iq :descr= && DESCRIZIONE=`echo $S | sed "s/^.*:descr=\([^:]*\):.*$/\\1/"`



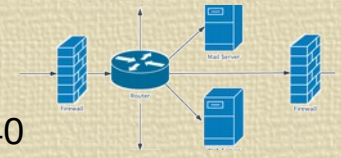
l'avvio e lo spegnimento delle macchine

- Lo script, interpretando il file di descrizione:
  - Avvia gli switch necessari
  - Nel caso trovi nella directory **~/umlnetsym** dell'utente la sottodirectory dell'esercizio con i file tar.bz2 delle vecchie directory ***/etc*** chiede all'utente se iniziare una nuova esecuzione o mantenere le configurazioni già predisposte
  - Predispone le directory con le cartelle ***/etc***, nuove o vecchie, le ***/modules*** e le home per ogni macchina
  - Produce la riga di comando necessaria per avviare le singole macchine virtuali.
- A questo punto basterebbe eseguire le righe di comando e mandarle in background, ma nel caso le macchine non consentissero più l'accesso all'utente, non ci sarebbe modo facile di spegnerle.
- Inoltre non potrei eseguire uno script al termine dell'esecuzione di ogni macchina per salvare i dati.



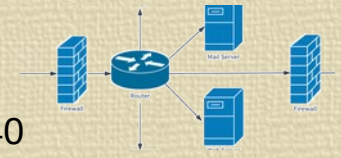
Soluzione dei problemi allo spegnimento delle macchine

- Per risolvere questi due problemi:
  - Viene lanciato uno script in background che esegue la macchina in modo che quando termina lo script riprende il controllo ed è in grado di salvare i dati.
  - Lo script lancia in background un secondo script che apre una finestra di dialogo con un bottone che consente all'utente di richiedere la terminazione forzata della macchina. Lo script conoscerà la macchina per cui viene lanciato e, se viene premuto il bottone di terminazione forzata, fa una *kill* sul processo della macchina virtuale (non dello script che l'ha lanciata) in modo che lo script possa intercettare la terminazione e fare le operazioni di salvataggio.
- Ogni volta che viene eseguito un codice di terminazione viene cancellato il file (visto prima) del *PID* della macchina. Se nella directory dell'esercizio non rimane più alcun file con il *PID* delle macchine, allora viene salvato il file dell'esercizio con tutti i dati definitivi.



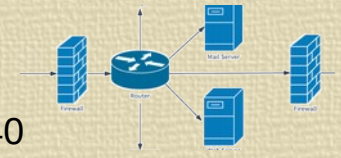
Ricaricare un esercizio

- Per correggere gli esercizi svolti è possibile analizzare le configurazioni ed i file salvati nelle home dei due utenti, per ogni macchina.
- Non sempre questo è comodo ed a volte non è nemmeno sufficiente.
- Sarebbe meglio poter riavviare la macchina dello studente e fare delle verifiche lanciando dei comandi.
- Per fare questo ho aggiunto una caratteristica: se nella directory dove è immagazzinato il file dell'esercizio da lanciare sono presenti i file di salvataggio di alcuni studenti, lo script apre una finestra con una lista dei file, estrapolandone il nome utente e la data di salvataggio e consente di scegliere se fare partire l'esercizio ex novo oppure ricaricare uno dei salvataggi presenti nella cartella.
- Nel caso si scelga di aprire un esercizio svolto, lo script, invece di inizializzare la directory **macchine**, vi scompatta il file di salvataggio, in modo che all'avvio la macchina si ritrovi nella stessa situazione in cui lo studente l'ha lasciata.



Considerazioni finali e test

- Lo strumento è abbastanza potente da fare esercizi anche complessi come ad esempio creare una *vpn* con **openvpn** tra due macchine o due reti.
- Contiene anche software per IPv6, ma la configurazione non è ancora pronta ed ho fatto solo esperimenti di base.
- A maggio scorso ho anche usato il prodotto in classe (o meglio in didattica a distanza), arrivando a fare una prova di laboratorio virtuale abbastanza complessa, su due diverse classi.
- Per facilitare il lavoro agli studenti ho prodotto un'immagine per Virtualbox con una distribuzione Debian Buster nella quale gli studenti hanno potuto lanciare l'emulatore ed eseguire gli esercizi
- La prova è andata abbastanza bene, anche se alcune macchine degli studenti hanno avuto qualche problema e la prova ha richiesto più tempo del previsto.
- Ho anche scoperto che il formato **xz** non è affidabile: più della metà degli esercizi salvati con **Xzip** sono arrivati corrotti del tutto o in parte quindi sono ritornato a **bzip2** che invece non ha dato problemi.



Dove trovarlo

- Il progetto umlnetsym è disponibile su sourceforge all'indirizzo:

  https://sourceforge.net/projects/umlnetsym/

- Il progetto nasce nel 2013, lo ritengo tuttora in beta ma è arrivato comunque alla sua quarta revisione maggiore