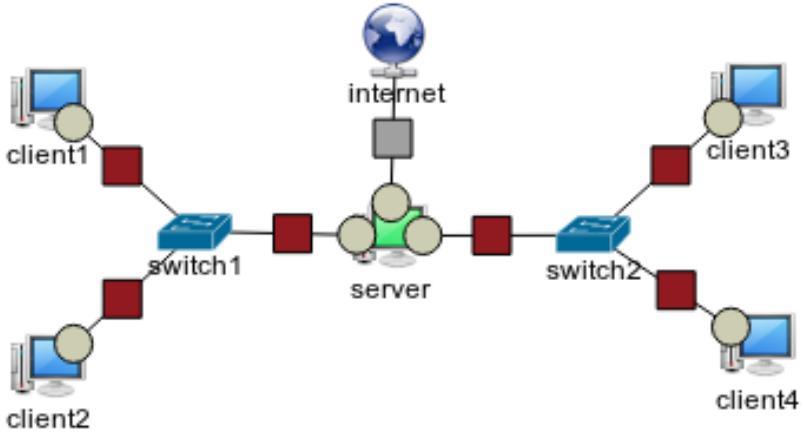
# ToMaTo - Programmable Devices

# ToMaTo, Topology Management Tool

ToMaTo is topology-oriented, i.e. users build topologies for their experiments.




## Devices

- produce and consume data
- can run software

## Connectors

- forward and manipulate data
- connect devices

## Three kinds of devices

- KVM devices
- OpenVZ devices
- Programmable devices

## Two kinds of connectors

- VPN networks
- External networks

# Programmable devices

Programmable devices run scripts written in Repy, a sandboxed Python dialect. Using these devices, networking data can be read and written as raw Ethernet packages.

## Difference between Repy and Python

- No global variables. Instead Repy has a dictionary **`mycontext`** that can be used to store global variables.
- No user input via **`input`** or **`raw_input`**
- Some Python builtins are not available. The most important are:
  - **`import`** and **`reload`**, Repy does not allow library loading
  - **`print`**, use **`echo("message")`** instead
  - **`input`**, **`raw_input`**, **`eval`** and **`execfile`**
  - **`lambda`** and **`yield`**
  - **`hasattr`**, **`getattr`** and **`setattr`**
- Parameters to the script are passed as **`callargs`**

# Script library: tomatolib

The library is part of the ToMaTo source code and located at Github. It contains the following:

- Implementations of common Internet protocols (some are stubs)
- Example Repy scripts
- Utility methods for Repy scripts
- Makefile to include libraries in scripts and build combined script

Library files can be found in the **`lib`** directrory and can be included like in the C programming language

```
#include <some_file/in_lib.repy>

def your_code(comes, here):
  ...
```

Build your final output script using these steps:

1. Put your code in the **`src`** directory
2. Call **`make`** in the base directory
3. Find your output script in the **`build`** directory

# Packet reading

## Reading from one device

```
packet = tuntap_read("eth0", timeout=None)
```

## Timeout values

- **`timeout=T`** waits at most for T seconds for a packet, otherwise returns None
- **`timeout=0.0`** returns packet (or None) immediately, no timeout
- **`timeout=None`** returns when a packet has been received, infinite timeout

## Reading from all devices at once

```
(dev, packet) = tuntap_read_any(timeout=None)
```

## Default reading loop

```
while True:
  try:
    (dev, packet) = tuntap_read_any(timeout=None)
    if packet:
      handle(dev, packet)
    except Exception, e:
      print_exc(e)
```

# Packet sending

## Sending on one device

```
1 tuntap_send("eth0", packet)
```

## Sending on all devices

```
1 for dev in tuntap_list():
2   send(dev, packet)
```

## Sending and receiving (simple switch example)

```
 1 #include <layer2/ethernet_proto.repy>
 2
 3 mac_table = {}
 4
 5 while True:
 6   (dev, packet) = tuntap_read_any(timeout=None)
 7   ether = ethernet_decode(packet)
 8   mac_table[ether.src] = dev
 9   dst = mac_table.get(ether.dst)
10   if dst:
11     tuntap_send(dst, packet)
12   else:
13     for dst in tuntap_list():
14       if dst != dev:
15         tuntap_send(dst, packet)
```

# Working with protocols - dissecting headers

## Packets are strings

- **`packet[index]`** is one character (one byte)
- **`len(packet)`** is the size of a packet
- **`packet[start:end]`** is a byte range from start to end-1
- **`ord(char)`** converts a character to a number
- **`chr(num)`** converts a number to a character

## Example: IP

```
# This is only the IP header

# Source and destination address are 4 bytes each starting at bytes 12 and 16
src = packet[12:16]
dst = packet[16:20]

# IP header length is last 4 bits of the first byte
ihl = ord(packet[0]) & 0x0f

# IP header length counts length in 4-byte blocks
# First 5 blocks are normal header, rest are option headers
options = []
for i in range(5, ihl):
  options.append(packet[4*i:4*i+4])

# After the header comes the payload
payload = packet[ihl*4:]
```

# Packing and unpacking binary data using struct

## Decoding binary data

```
(byte1, byte2, int1, int2) = struct.unpack("!BBHH", packet[0:6])
```

## Encoding binary data

```
packet = struct.pack("!BBHH", byte1, byte2, int1, int2)
```

## Struct codes

- **!** means network byte order, must be first character
- Each character stands for one field
  - **B** is a number that takes 1 byte (**B**yte)
  - **H** is a number that takes 2 bytes (**H**alf-integer)
  - **I** is a number that takes 4 bytes (**I**nteger)
  - **Q** is a number that takes 8 bytes
- Uppercase is unsigned, lowercase is signed

See the python struct documentation for more info.

# Tips & Tricks

***Code sharing:*** *Share your code with others on Github to improve the library and to get feedback.*

***Performance:*** *Code that uses struct to convert binary data to numbers is a faster but the fastest code avoids convertions.*

***Exception handling:*** *Wrap the per-packet code in a try-except block, otherwise the script will abort on an error.*

***80-20 rule:*** *80% of the functionality of a protocol is implemented in 20% of its code, and vice versa. Most protocols have optional features that are not needed in most cases.*

***Arguments:*** *Scripts can take arguments, no need to write different scripts just to have different addresses.*

# Obtaining and contributing

## How to get ToMaTo

ToMaTo is Open-Source! It can be simply downloaded from the Github page. There is also a step-by-step tutorial on how to setup ToMaTo in a testbed. ToMaTo includes some nice features that make it pretty easy to install it in an experimental facility:

- All components packaged for debian (updates come automatically)
- Multiple authentication plugins: LDAP, htaccess, SQL-Database, Planet-Lab, ...
- Automatic checks and problem reports for the ToMaTo hosts

## How to contribute to ToMaTo

As an Open-Source project ToMaTo is open for hints and contributions.

- Github offers an easy way to fork the project and offer contributions as pull requests
- The wiki is publically editable so everyone can help by adding to the documentation
- The issue tracking system can be used for bug reports and feature requests