net2o: Reinventing the Internet

Bernd Paysan

YBTI session, TUM, Garching, 8-3

Outline

- net2o in a nutshell
- Topology
  - Low–Overhead Packet Format
- Encryption
  - Key Exchange
  - Symmetric Crypto
- Flow Control
- Commands
- Distributed Data
- Applications
  - Apps in a Sandbox
  - API Basics

net2o in a nutshell

net2o consists of the following 6 layers (implemented bottom up):

2. Path switched packets with 2^n size writing into shared memory buffers
3. Ephemeral key exchange and signatures with Ed25519, symmetric authenticated encryption+hash+prng with Keccak
4. Timing driven delay minimizing flow control
5. Stack–oriented tokenized command language
6. Distributed data (files) and distributed metadata (prefix hash trie)
7. Apps in a sandboxed environment for displaying content

net2o in a nutshell

net2o consists of the following 6 layers (implemented bottom up):

2. Path switched packets with 2^n size writing into shared memory buffers
3. Ephemeral key exchange and signatures with Ed25519, symmetric authenticated encryption+hash+prng with Keccak
4. Timing driven delay minimizing flow control
5. Stack–oriented tokenized command language
6. Distributed data (files) and distributed metadata (prefix hash trie)
7. Apps in a sandboxed environment for displaying content

net2o in a nutshell

net2o consists of the following 6 layers (implemented bottom up):

2. Path switched packets with 2^n size writing into shared memory buffers
3. Ephemeral key exchange and signatures with Ed25519, symmetric authenticated encryption+hash+prng with Keccak
4. Timing driven delay minimizing flow control
5. Stack–oriented tokenized command language
6. Distributed data (files) and distributed metadata (prefix hash trie)
7. Apps in a sandboxed environment for displaying content

net2o in a nutshell

net2o consists of the following 6 layers (implemented bottom up):

2. Path switched packets with 2^n size writing into shared memory buffers
3. Ephemeral key exchange and signatures with Ed25519, symmetric authenticated encryption+hash+prng with Keccak
4. Timing driven delay minimizing flow control
5. Stack–oriented tokenized command language
6. Distributed data (files) and distributed metadata (prefix hash trie)
7. Apps in a sandboxed environment for displaying content

net2o in a nutshell

net2o consists of the following 6 layers (implemented bottom up):

2. Path switched packets with 2^n size writing into shared memory buffers
3. Ephemeral key exchange and signatures with Ed25519, symmetric authenticated encryption+hash+prng with Keccak
4. Timing driven delay minimizing flow control
5. Stack–oriented tokenized command language
6. Distributed data (files) and distributed metadata (prefix hash trie)
7. Apps in a sandboxed environment for displaying content

net2o in a nutshell

net2o consists of the following 6 layers (implemented bottom up):

2. Path switched packets with 2^n size writing into shared memory buffers
3. Ephemeral key exchange and signatures with Ed25519, symmetric authenticated encryption+hash+prng with Keccak
4. Timing driven delay minimizing flow control
5. Stack–oriented tokenized command language
6. Distributed data (files) and distributed metadata (prefix hash trie)
7. Apps in a sandboxed environment for displaying content

net2o in a nutshell

net2o consists of the following 6 layers (implemented bottom up):

2. Path switched packets with 2^n size writing into shared memory buffers
3. Ephemeral key exchange and signatures with Ed25519, symmetric authenticated encryption+hash+prng with Keccak
4. Timing driven delay minimizing flow control
5. Stack–oriented tokenized command language
6. Distributed data (files) and distributed metadata (prefix hash trie)
7. Apps in a sandboxed environment for displaying content

Objectives

net2o's design objectives are

- lightweight, fast, scalable
- easy to implement
- secure
- media capable
- works as overlay on current networks (UDP/IP), but can replace the entire stack

Objectives

net2o's design objectives are

- lightweight, fast, scalable
- easy to implement
- secure
- media capable
- works as overlay on current networks (UDP/IP), but can replace the entire stack

Objectives

net2o's design objectives are

- lightweight, fast, scalable
- easy to implement
- secure
- media capable
- works as overlay on current networks (UDP/IP), but can replace the entire stack

Objectives

net2o's design objectives are

- lightweight, fast, scalable
- easy to implement
- secure
- media capable
- works as overlay on current networks (UDP/IP), but can replace the entire stack

Objectives

net2o's design objectives are

- lightweight, fast, scalable
- easy to implement
- secure
- media capable
- works as overlay on current networks (UDP/IP), but can replace the entire stack

Objectives

net2o's design objectives are

- lightweight, fast, scalable
- easy to implement
- secure
- media capable
- works as overlay on current networks (UDP/IP), but can replace the entire stack

Switching Packets, Routing Connections

- Switches are faster and easier to implement than routers — LANs (Ethernet) and backbones (MPLS) already use switching; use the concept of MPLS label stacks to use switching everywhere
- Routing then is a combination of destination resolution and routing calculation (destination path lookup)

Path Switching

- Take first *n* bits of path field and select destination
- Shift target address by *n*
- Insert bit-reversed source into the rear end of the path field to mark the way back

- The receiver bit–flips the path field, and gets the return address
- Easy handover possible

Switching Packets, Routing Connections

- Switches are faster and easier to implement than routers — LANs (Ethernet) and backbones (MPLS) already use switching; use the concept of MPLS label stacks to use switching everywhere
- Routing then is a combination of destination resolution and routing calculation (destination path lookup)

Path Switching

- Take first *n* bits of path field and select destination
- Shift target address by *n*
- Insert bit-reversed source into the rear end of the path field to mark the way back

- The receiver bit–flips the path field, and gets the return address
- Easy handover possible

Switching Packets, Routing Connections

- Switches are faster and easier to implement than routers — LANs (Ethernet) and backbones (MPLS) already use switching; use the concept of MPLS label stacks to use switching everywhere
- Routing then is a combination of destination resolution and routing calculation (destination path lookup)

Path Switching

- Take first *n* bits of path field and select destination
- Shift target address by *n*
- Insert bit-reversed source into the rear end of the path field to mark the way back

- The receiver bit–flips the path field, and gets the return address
- Easy handover possible

Switching Packets, Routing Connections

- Switches are faster and easier to implement than routers — LANs (Ethernet) and backbones (MPLS) already use switching; use the concept of MPLS label stacks to use switching everywhere
- Routing then is a combination of destination resolution and routing calculation (destination path lookup)

Path Switching

- Take first *n* bits of path field and select destination
- Shift target address by *n*
- Insert bit-reversed source into the rear end of the path field to mark the way back

- The receiver bit–flips the path field, and gets the return address
- Easy handover possible

Switching Packets, Routing Connections

- Switches are faster and easier to implement than routers — LANs (Ethernet) and backbones (MPLS) already use switching; use the concept of MPLS label stacks to use switching everywhere
- Routing then is a combination of destination resolution and routing calculation (destination path lookup)

Path Switching

- Take first *n* bits of path field and select destination
- Shift target address by *n*
- Insert bit-reversed source into the rear end of the path field to mark the way back

- The receiver bit–flips the path field, and gets the return address
- Easy handover possible

Switching Packets, Routing Connections

- Switches are faster and easier to implement than routers — LANs (Ethernet) and backbones (MPLS) already use switching; use the concept of MPLS label stacks to use switching everywhere
- Routing then is a combination of destination resolution and routing calculation (destination path lookup)

Path Switching

- Take first *n* bits of path field and select destination
- Shift target address by *n*
- Insert bit-reversed source into the rear end of the path field to mark the way back

- The receiver bit–flips the path field, and gets the return address
- Easy handover possible

Switching Packets, Routing Connections

- Switches are faster and easier to implement than routers — LANs (Ethernet) and backbones (MPLS) already use switching; use the concept of MPLS label stacks to use switching everywhere
- Routing then is a combination of destination resolution and routing calculation (destination path lookup)

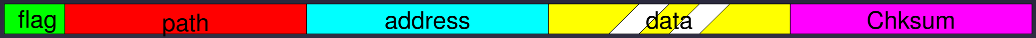
Path Switching

- Take first n bits of path field and select destination
- Shift target address by n
- Insert bit-reversed source into the rear end of the path field to mark the way back

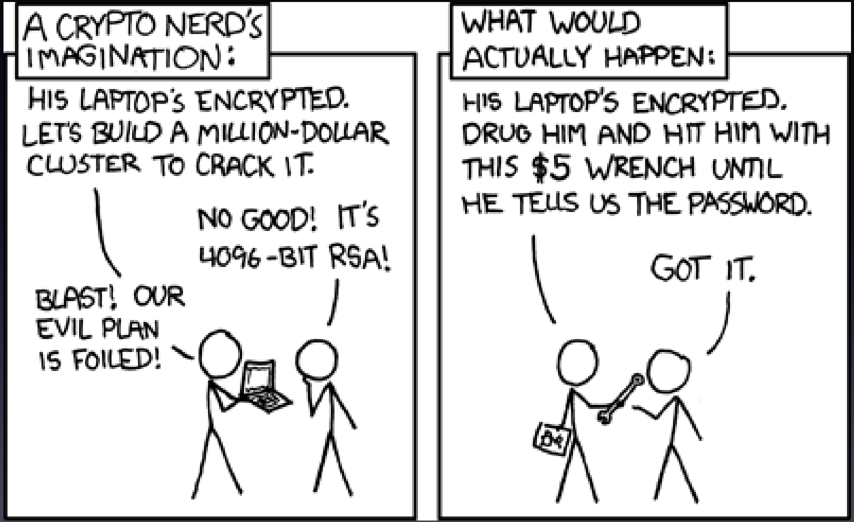
- The receiver bit–flips the path field, and gets the return address
- Easy handover possible

Packet Format

	Bytes	*Comment*
Flags	2	priority, length, flow control flags
Path	16	Internet 1.0 terminology: “address”
Address	8	address in memory, ≈port+sequence number
Data	$64 * 2^{0..15}$	up to 2MB packet size, enough for the next 40 years
Chksum	16	cryptographic checksum



Security: Indirect Attacks are Cheaper



Key Exchange

ECC Elliptic Curve Cryptography has still only a generic attack (i.e. can be considered "unscratched", as the attack uses a fundamental property of the problem), and therefore 256 bit keys (32 bytes) have a strength of 128 bits

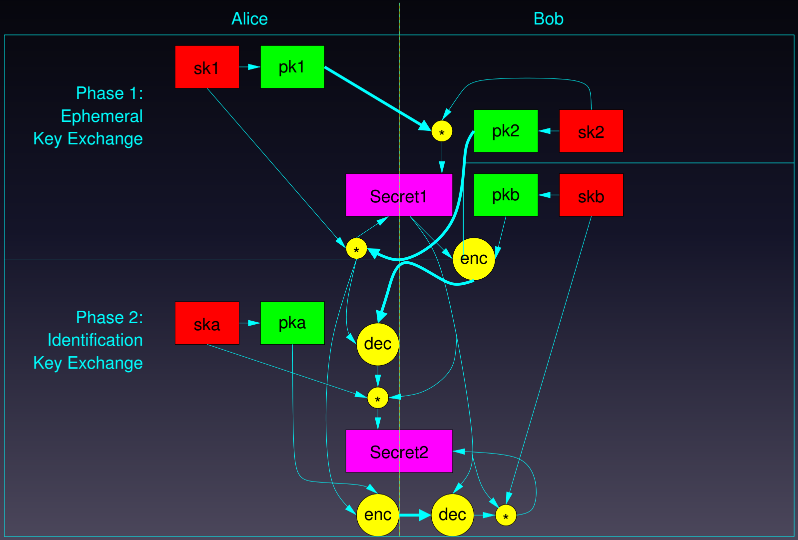
Therefore the choice now is Ed25519, a variant of Curve25519 from Dan Bernstein that supports signatures, too. This is a curve where the parameters are of high quality.

Key Exchange

ECC Elliptic Curve Cryptography has still only a generic attack (i.e. can be considered “unscratched”, as the attack uses a fundamental property of the problem), and therefore 256 bit keys (32 bytes) have a strength of 128 bits

Therefore the choice now is Ed25519, a variant of Curve25519 from Dan Bernstein that supports signatures, too. This is a curve where the parameters are of high quality.

Ephemeral Key Exchange+Validation



Symmetric Crypto: Keccak

Keccak used for the following reasons:

- Good cryptanalysis
- Keccak in duplex mode provides perfect side–channel protected AEAD operation (no constant key to snoop)
- Strength >256 bits: very good security margin
- Keccak is a universal crypto primitive (hash+encrypt+authenticate)
- Keccak is both NIST–approved and (still) NSA–independent. I use Keccak with $r = 1024$ and capacity $c = 576$ as suggested by the Keccak authors.

Symmetric Crypto: Keccak

Keccak used for the following reasons:

- Good cryptanalysis
- Keccak in duplex mode provides perfect side–channel protected AEAD operation (no constant key to snoop)
- Strength >256 bits: very good security margin
- Keccak is a universal crypto primitive (hash+encrypt+authenticate)
- Keccak is both NIST–approved and (still) NSA–independent. I use Keccak with $r = 1024$ and capacity $c = 576$ as suggested by the Keccak authors.

Symmetric Crypto: Keccak

Keccak used for the following reasons:

- Good cryptanalysis
- Keccak in duplex mode provides perfect side–channel protected AEAD operation (no constant key to snoop)
- Strength >256 bits: very good security margin
- Keccak is a universal crypto primitive (hash+encrypt+authenticate)
- Keccak is both NIST–approved and (still) NSA–independent. I use Keccak with $r = 1024$ and capacity $c = 576$ as suggested by the Keccak authors.

Symmetric Crypto: Keccak

Keccak used for the following reasons:

- Good cryptanalysis
- Keccak in duplex mode provides perfect side–channel protected AEAD operation (no constant key to snoop)
- Strength >256 bits: very good security margin
- Keccak is a universal crypto primitive (hash+encrypt+authenticate)
- Keccak is both NIST–approved and (still) NSA–independent. I use Keccak with $r = 1024$ and capacity $c = 576$ as suggested by the Keccak authors.

Symmetric Crypto: Keccak

Keccak used for the following reasons:

- Good cryptanalysis
- Keccak in duplex mode provides perfect side–channel protected AEAD operation (no constant key to snoop)
- Strength >256 bits: very good security margin
- Keccak is a universal crypto primitive (hash+encrypt+authenticate)
- Keccak is both NIST–approved and (still) NSA–independent. I use Keccak with $r = 1024$ and capacity $c = 576$ as suggested by the Keccak authors.

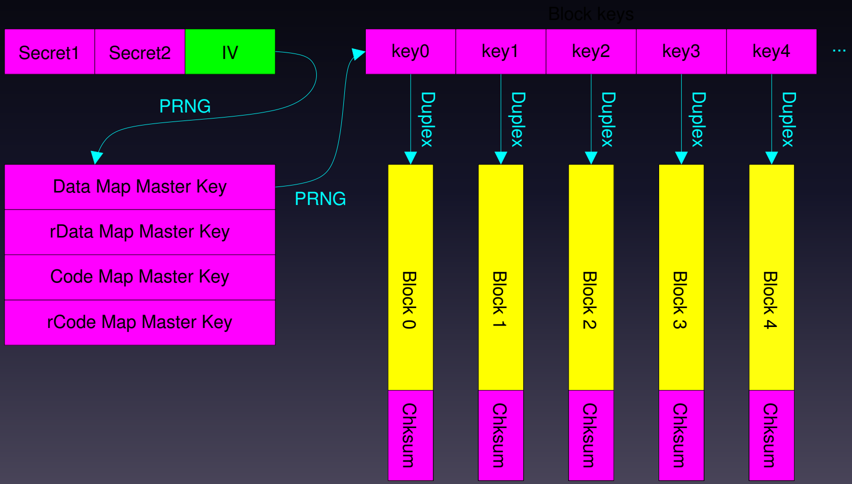
Symmetric Crypto: Keccak

Keccak used for the following reasons:

- Good cryptanalysis
- Keccak in duplex mode provides perfect side–channel protected AEAD operation (no constant key to snoop)
- Strength $>$256 bits: very good security margin
- Keccak is a universal crypto primitive (hash+encrypt+authenticate)
- Keccak is both NIST–approved and (still) NSA–independent. I use Keccak with $r = 1024$ and capacity $c = 576$ as suggested by the Keccak authors.

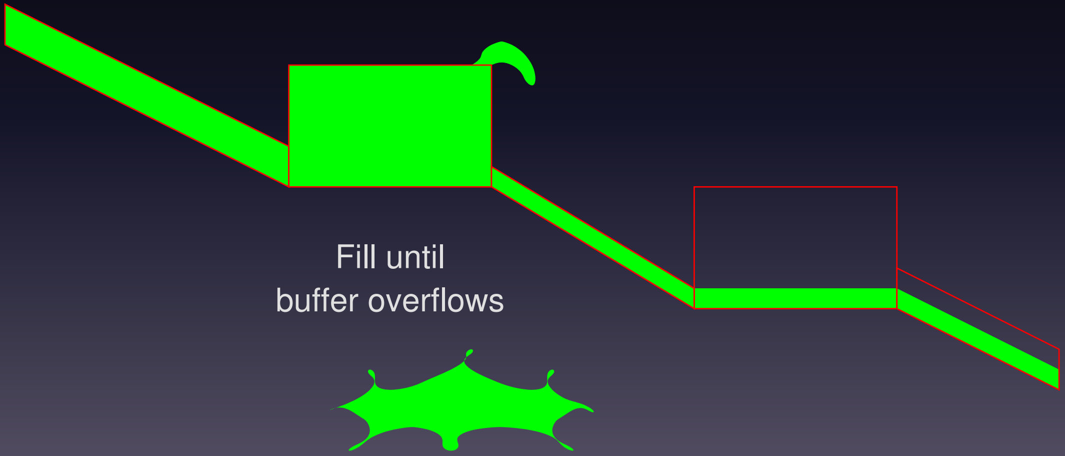
Key Usage

All keys are one–time–use only!



Flow Control (Broken)

- TCP fills the buffer, until a packet has to be dropped, instead of reducing rate before. Name of the symptom: "Buffer bloat". But buffering is essential for good network performance.



Alternatives?

- LEDBAT tries to achieve a low, constant delay: Works, but not good on fairness
- CurveCP's flow control is still "a lot of research"
- Therefore, something new has to be done

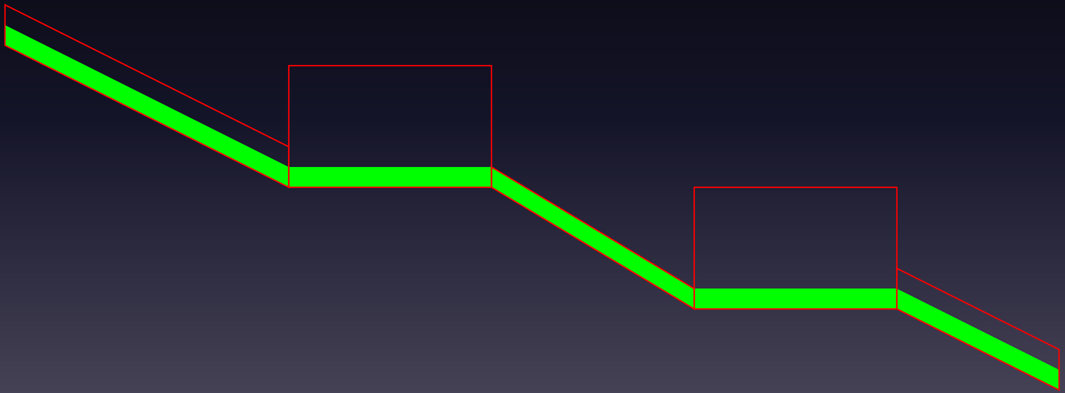


Figure : That's how proper flow control should look like

Alternatives?

- LEDBAT tries to achieve a low, constant delay: Works, but not good on fairness
- CurveCP's flow control is still "a lot of research"
- Therefore, something new has to be done

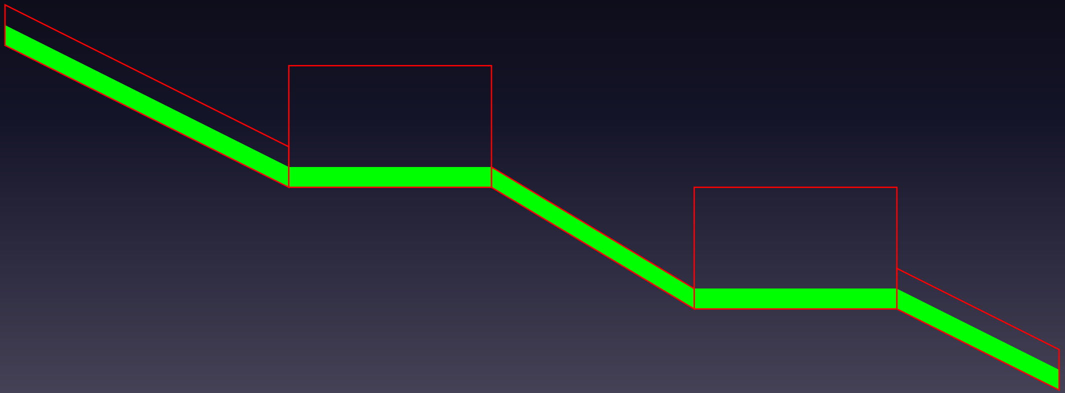


Figure : That's how proper flow control should look like

Alternatives?

- LEDBAT tries to achieve a low, constant delay: Works, but not good on fairness
- CurveCP's flow control is still "a lot of research"
- Therefore, something new has to be done

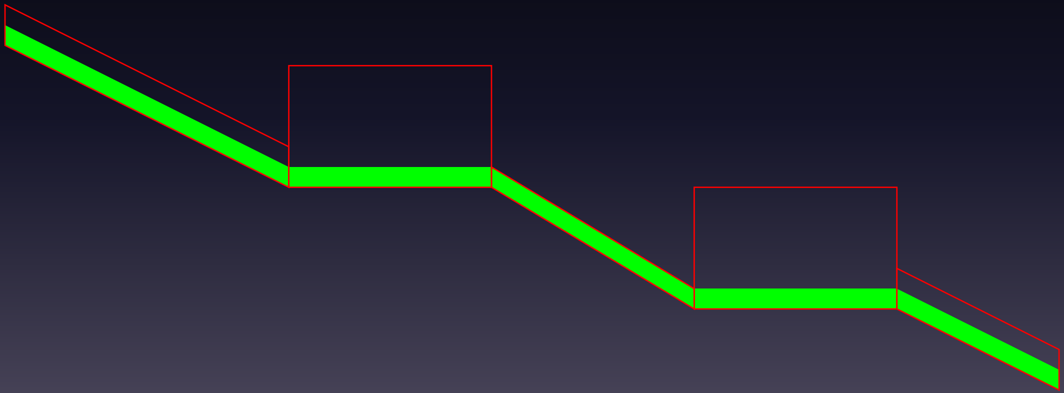


Figure : That's how proper flow control should look like

net2o Flow Control

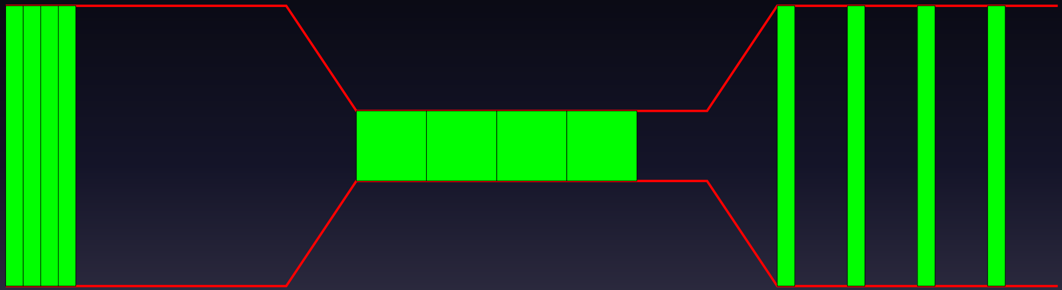


Figure : Measure the bottleneck using a burst of packets

Client Measures, Server Sets Rate

Client recores the *time* of the first and last packet in a burst, and calculates the achieved rate for received packets, extrapolating to the achievable rate including the dropped packets. This results in the requested *rate*.

$$rate := \Delta t * \frac{burstlen}{packets}$$

Server would simply use this rate

Client Measures, Server Sets Rate

Client recores the *time* of the first and last packet in a burst, and calculates the achieved rate for received packets, extrapolating to the achievable rate including the dropped packets. This results in the requested *rate*.

$$rate := \Delta t * \frac{burstlen}{packets}$$

Server would simply use this rate

Fairness

Fairness means that concurrent connections achieve about the same data rate, sharing the same line in a fair way.

- Ideally, a router/switch would schedule buffered packets round–robin, giving each connection a fair share of the bandwidth. That would change the calculated rate appropriately, and also be a big relief for current TCP buffer bloat symptoms, as each connection would have its private buffer to fill up.
- Unfortunately, routers use a single FIFO policy for all connections
- Finding a sufficiently stable algorithm to provide fairness
- We want to adopt to new situations as fast as possible, there's no point in anything slow. Especially on wireless connections, achievable rate changes are not only related to traffic.

Fairness

Fairness means that concurrent connections achieve about the same data rate, sharing the same line in a fair way.

- Ideally, a router/switch would schedule buffered packets round–robin, giving each connection a fair share of the bandwidth. That would change the calculated rate appropriately, and also be a big relief for current TCP buffer bloat symptoms, as each connection would have its private buffer to fill up.
- Unfortunately, routers use a single FIFO policy for all connections
- Finding a sufficiently stable algorithm to provide fairness
- We want to adopt to new situations as fast as possible, there's no point in anything slow. Especially on wireless connections, achievable rate changes are not only related to traffic.

Fairness

Fairness means that concurrent connections achieve about the same data rate, sharing the same line in a fair way.

- Ideally, a router/switch would schedule buffered packets round–robin, giving each connection a fair share of the bandwidth. That would change the calculated rate appropriately, and also be a big relief for current TCP buffer bloat symptoms, as each connection would have its private buffer to fill up.
- Unfortunately, routers use a single FIFO policy for all connections
- Finding a sufficiently stable algorithm to provide fairness
- We want to adopt to new situations as fast as possible, there's no point in anything slow. Especially on wireless connections, achievable rate changes are not only related to traffic.

Fairness

Fairness means that concurrent connections achieve about the same data rate, sharing the same line in a fair way.

- Ideally, a router/switch would schedule buffered packets round–robin, giving each connection a fair share of the bandwidth. That would change the calculated rate appropriately, and also be a big relief for current TCP buffer bloat symptoms, as each connection would have its private buffer to fill up.
- Unfortunately, routers use a single FIFO policy for all connections
- Finding a sufficiently stable algorithm to provide fairness
- We want to adopt to new situations as fast as possible, there's no point in anything slow. Especially on wireless connections, achievable rate changes are not only related to traffic.

Fairness

Fairness means that concurrent connections achieve about the same data rate, sharing the same line in a fair way.

- Ideally, a router/switch would schedule buffered packets round–robin, giving each connection a fair share of the bandwidth. That would change the calculated rate appropriately, and also be a big relief for current TCP buffer bloat symptoms, as each connection would have its private buffer to fill up.
- Unfortunately, routers use a single FIFO policy for all connections
- Finding a sufficiently stable algorithm to provide fairness
- We want to adopt to new situations as fast as possible, there's no point in anything slow. Especially on wireless connections, achievable rate changes are not only related to traffic.

net2o Flow Control — Fair Router

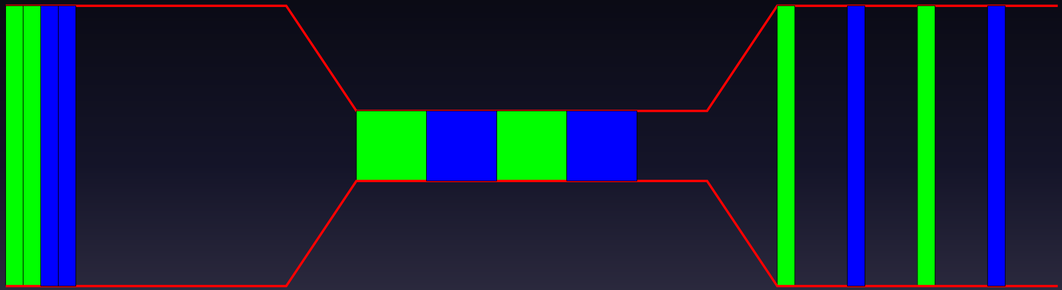


Figure : Fair queuing results in correct measurement of available bandwidth

net2o Flow Control — FIFO Router

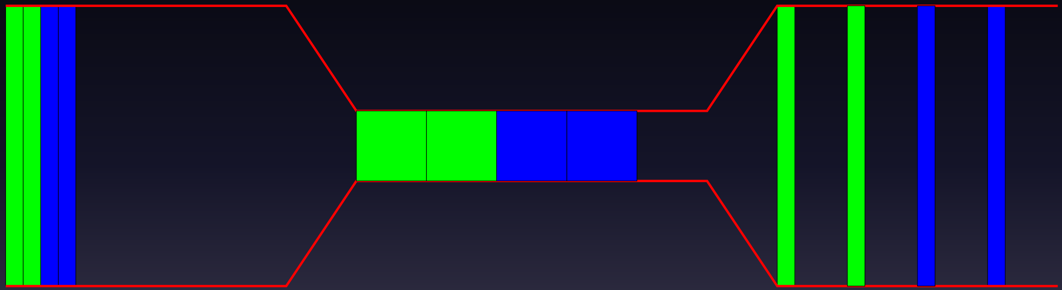


Figure : Unfair FIFO queuing results in twice the available bandwidth calculated

Fairness I

- To improve stability of unfair queued packets, we need to improve that P regulator (proportional to measured rate) to a full PID regulator
- The integral part is the accumulated slack (in the buffer), which we want to keep low, and the D part is growing/reducing this slack from one measurement to the next
- We use both parts to decrease the sending rate, and thereby achieve better fairness
- The I part is used to exponentially lengthen the rate Δt with increasing slack up to a maximum factor of 16.

$$s_{exp} = 2^{\frac{slack}{T}} \quad \text{where } T = \max(10ms, \max(slacks))$$

Fairness I

- To improve stability of unfair queued packets, we need to improve that P regulator (proportional to measured rate) to a full PID regulator
- The integral part is the accumulated slack (in the buffer), which we want to keep low, and the D part is growing/reducing this slack from one measurement to the next
- We use both parts to decrease the sending rate, and thereby achieve better fairness
- The I part is used to exponentially lengthen the rate Δt with increasing slack up to a maximum factor of 16.

$$s_{exp} = 2^{\frac{slack}{T}} \quad \text{where } T = \max(10ms, \max(slacks))$$

Fairness I

- To improve stability of unfair queued packets, we need to improve that P regulator (proportional to measured rate) to a full PID regulator
- The integral part is the accumulated slack (in the buffer), which we want to keep low, and the D part is growing/reducing this slack from one measurement to the next
- We use both parts to decrease the sending rate, and thereby achieve better fairness
- The I part is used to exponentially lengthen the rate Δt with increasing slack up to a maximum factor of 16.

$$s_{exp} = 2^{\frac{slack}{T}} \quad \text{where } T = \max(10ms, \max(slacks))$$

Fairness I

- To improve stability of unfair queued packets, we need to improve that P regulator (proportional to measured rate) to a full PID regulator
- The integral part is the accumulated slack (in the buffer), which we want to keep low, and the D part is growing/reducing this slack from one measurement to the next
- We use both parts to decrease the sending rate, and thereby achieve better fairness
- The I part is used to exponentially lengthen the rate Δt with increasing slack up to a maximum factor of 16.

$$s_{exp} = 2^{\frac{slack}{T}} \quad \text{where } T = \max(10ms, \max(slacks))$$

Fairness I

- To improve stability of unfair queued packets, we need to improve that P regulator (proportional to measured rate) to a full PID regulator
- The integral part is the accumulated slack (in the buffer), which we want to keep low, and the D part is growing/reducing this slack from one measurement to the next
- We use both parts to decrease the sending rate, and thereby achieve better fairness
- The I part is used to exponentially lengthen the rate Δt with increasing slack up to a maximum factor of 16.

$$s_{exp} = 2^{\frac{slack}{T}} \quad \text{where } T = \max(10ms, \max(slacks))$$

Fairness D

- To measure the differential term, we measure how much the slack grows (a Δt value) from the first to the last burst we do for one measurement cycle (4 bursts by default, first packet to first packet of each burst)
- This is multiplied by the total packets in flight (head of the sender queue vs. acknowledged packet), divided by the packets within the measured interval
- A low–pass filter is applied to the obtained D to prevent from speeding up too fast, with one round trip delay as time constant
- $\max(slacks)/10ms$ is used to determine how aggressive this algorithm is
- Add the obtained Δt both to the rate's Δt for one burst sequence and wait that time before starting the next burst sequence

Fairness D

- To measure the differential term, we measure how much the slack grows (a Δt value) from the first to the last burst we do for one measurement cycle (4 bursts by default, first packet to first packet of each burst)
- This is multiplied by the total packets in flight (head of the sender queue vs. acknowledged packet), divided by the packets within the measured interval
- A low–pass filter is applied to the obtained D to prevent from speeding up too fast, with one round trip delay as time constant
- $\max(slacks)/10ms$ is used to determine how aggressive this algorithm is
- Add the obtained Δt both to the rate's Δt for one burst sequence and wait that time before starting the next burst sequence

Fairness D

- To measure the differential term, we measure how much the slack grows (a Δt value) from the first to the last burst we do for one measurement cycle (4 bursts by default, first packet to first packet of each burst)
- This is multiplied by the total packets in flight (head of the sender queue vs. acknowledged packet), divided by the packets within the measured interval
- A low–pass filter is applied to the obtained D to prevent from speeding up too fast, with one round trip delay as time constant
- $\max(slacks)/10ms$ is used to determine how aggressive this algorithm is
- Add the obtained Δt both to the rate's Δt for one burst sequence and wait that time before starting the next burst sequence

Fairness D

- To measure the differential term, we measure how much the slack grows (a Δt value) from the first to the last burst we do for one measurement cycle (4 bursts by default, first packet to first packet of each burst)
- This is multiplied by the total packets in flight (head of the sender queue vs. acknowledged packet), divided by the packets within the measured interval
- A low–pass filter is applied to the obtained D to prevent from speeding up too fast, with one round trip delay as time constant
- $\max(\mathit{slacks})/10\mathit{ms}$ is used to determine how aggressive this algorithm is
- Add the obtained Δt both to the rate's Δt for one burst sequence and wait that time before starting the next burst sequence

Fairness D

- To measure the differential term, we measure how much the slack grows (a Δt value) from the first to the last burst we do for one measurement cycle (4 bursts by default, first packet to first packet of each burst)
- This is multiplied by the total packets in flight (head of the sender queue vs. acknowledged packet), divided by the packets within the measured interval
- A low–pass filter is applied to the obtained D to prevent from speeding up too fast, with one round trip delay as time constant
- $\max(slacks)/10ms$ is used to determine how aggressive this algorithm is
- Add the obtained Δt both to the rate's Δt for one burst sequence and wait that time before starting the next burst sequence.

VDSL

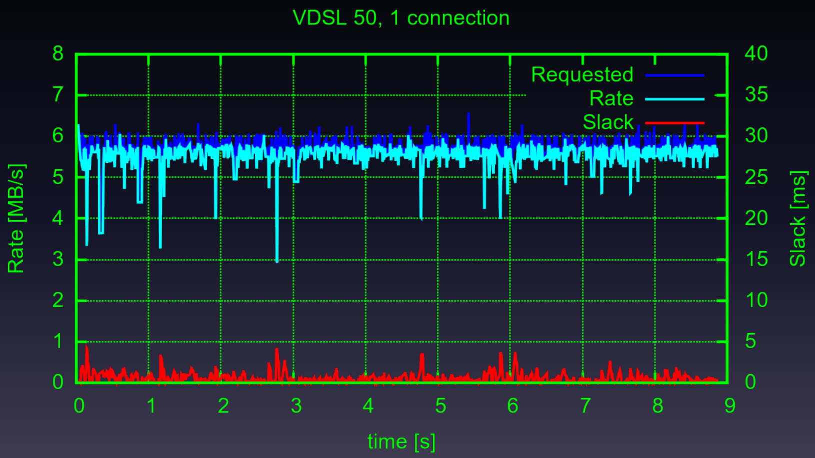


Figure : One connection on a VDSL–50 line

VDSL, Congestion

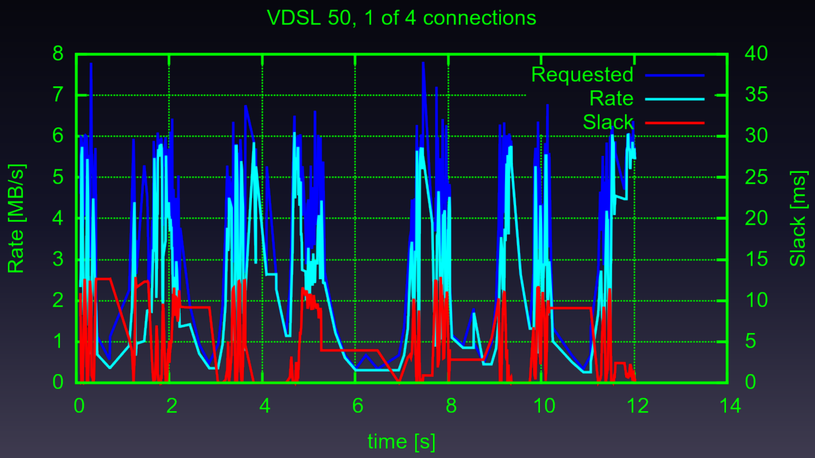


Figure : One of four connections on a VDSL–50 line

Unreliable Air Cable (WLAN)

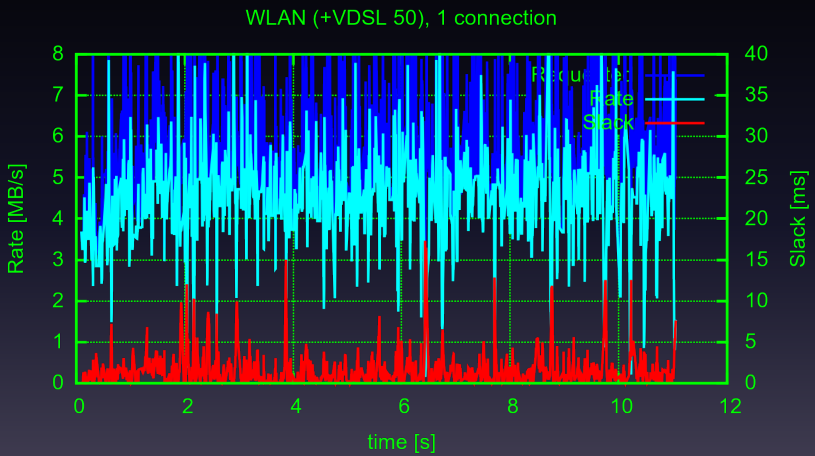


Figure : Single connection using WLAN

Unreliable Air Cable, Congestion

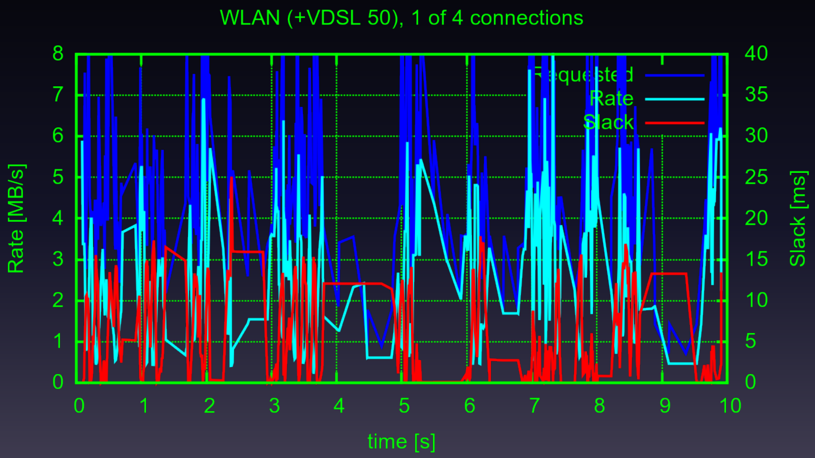


Figure : One of four connections using WLAN

LAN, 1GBE

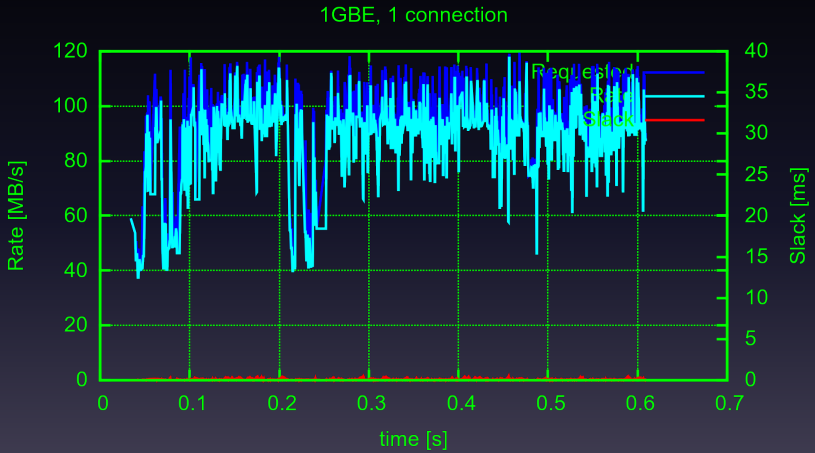


Figure : Single connection using 1GBE

LAN 1GBE, Congestion (4 servers)

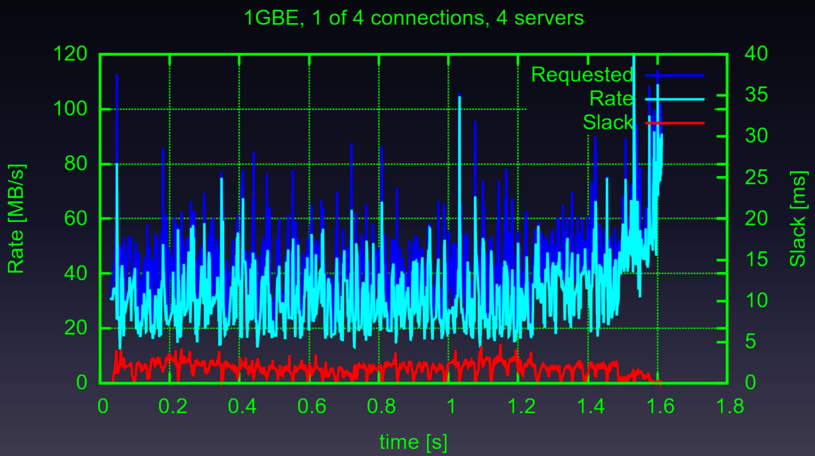


Figure : One of four connections using 1GBE

LAN 1GBE, Congestion (1 server)

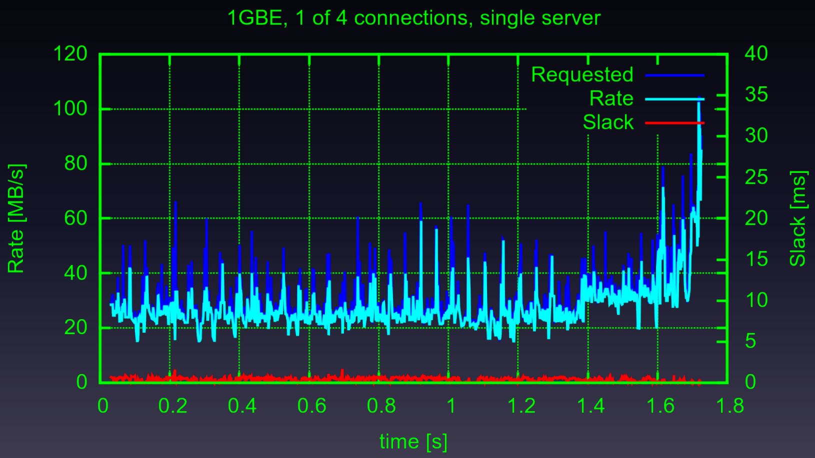


Figure : One of four connections using 1GBE, fair queuing

Data and Commands

- Data of several files/streams can be transferred interleaving, so a single connection can do multiple things in parallel
- Commands are send in command blocks, i.e. there is not just one command per block, but a sequence of commands!
- Commands are encoded like protobuf, i.e. 7 bits per byte, and if the MSB of the byte is 1, there's another byte to follow (allowing arbitrary many commands)
- The command "machine" is a stack architecture.

Data and Commands

- Data of several files/streams can be transferred interleaving, so a single connection can do multiple things in parallel
- Commands are send in command blocks, i.e. there is not just one command per block, but a sequence of commands!
- Commands are encoded like protobuf, i.e. 7 bits per byte, and if the MSB of the byte is 1, there's another byte to follow (allowing arbitrary many commands)
- The command "machine" is a stack architecture.

Data and Commands

- Data of several files/streams can be transferred interleaving, so a single connection can do multiple things in parallel
- Commands are send in command blocks, i.e. there is not just one command per block, but a sequence of commands!
- Commands are encoded like protobuf, i.e. 7 bits per byte, and if the MSB of the byte is 1, there's another byte to follow (allowing arbitrary many commands)
- The command "machine" is a stack architecture.

Data and Commands

- Data of several files/streams can be transferred interleaving, so a single connection can do multiple things in parallel
- Commands are send in command blocks, i.e. there is not just one command per block, but a sequence of commands!
- Commands are encoded like protobuf, i.e. 7 bits per byte, and if the MSB of the byte is 1, there's another byte to follow (allowing arbitrary many commands)
- The command "machine" is a stack architecture.

Example: Download three files

```
net2o-code
"Download test" $, type cr ( see-me )
get-ip $400 blocksize! $400 blockalign! stat( request-stats )
"net2o.fs" 0 lit, 0 lit, open-tracked-file
"data/2011-05-13_11-26-57-small.jpg" 0 lit, 1 lit, open-tracked-file
"data/2011-05-20_17-01-12-small.jpg" 0 lit, 2 lit, open-tracked-file
gen-total slurp-all-tracked-blocks send-chunks
0 lit, tag-reply
end-code
```

Distributed Data

- Following the “everything is a file” principle, every data object is a file
- Data objects are accessed by their hash. The associated metadata are “tags”
- Metadata is organized as a distributed prefix hash tree
- Efficient distribution of data is important!

Distributed Data

- Following the “everything is a file” principle, every data object is a file
- Data objects are accessed by their hash. The associated metadata are “tags”
- Metadata is organized as a distributed prefix hash tree
- Efficient distribution of data is important!

Distributed Data

- Following the “everything is a file” principle, every data object is a file
- Data objects are accessed by their hash. The associated metadata are “tags”
- Metadata is organized as a distributed prefix hash tree
- Efficient distribution of data is important!

Distributed Data

- Following the “everything is a file” principle, every data object is a file
- Data objects are accessed by their hash. The associated metadata are “tags”
- Metadata is organized as a distributed prefix hash tree
- Efficient distribution of data is important!

Tree Distribution Network

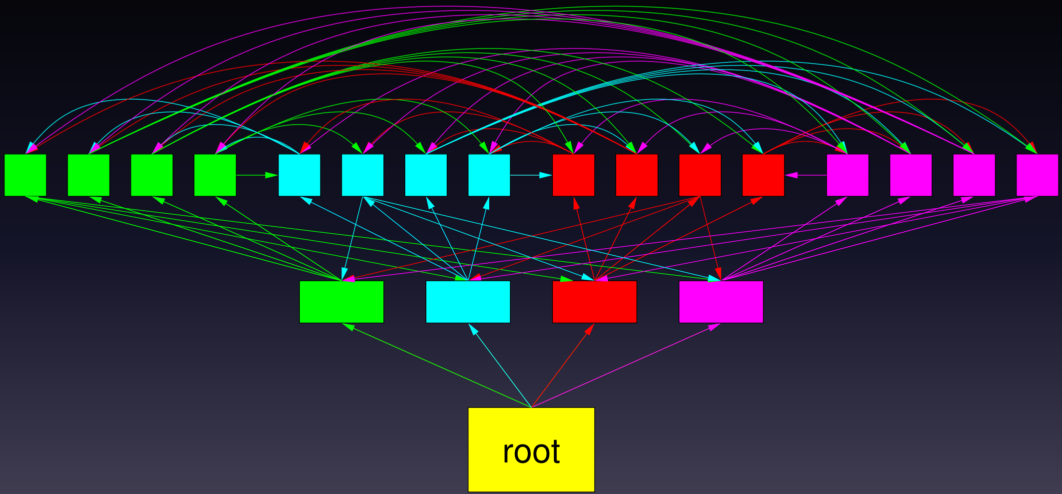


Figure : Avalanche distribution with quad–tree of depth 2

Possible Performance

- Trees with a bigger base reduce latency. Example: To transfer a Justin Bieber tweet to 50 million followers, a binary tree needs 25.5 hops on average, a quad–tree 12.8 hops, and an oct–tree 8.5 hops.
- A typical domestic (inside e.g. Germany) hop–to–hop time is just 20ms. International hops can be in the order of 250ms. Assuming there is only one international hop in the chain, the latency to distribute Justin Bieber's babbling is typically just 500ms in a quad–tree.

  Rule of thumb: *bandwidth* = *latency*, i.e. if it takes 20ms from hop to hop, each node should replicate data for 20ms — if we make the tree wider, the linear effort of replicating data will dominate transfer time, if we make the tree more narrow, the hop–to–hop time will dominate.
- The tree–like graph greatly reduces the number of nodes to know

Possible Performance

- Trees with a bigger base reduce latency. Example: To transfer a Justin Bieber tweet to 50 million followers, a binary tree needs 25.5 hops on average, a quad–tree 12.8 hops, and an oct–tree 8.5 hops.
- A typical domestic (inside e.g. Germany) hop–to–hop time is just 20ms. International hops can be in the order of 250ms. Assuming there is only one international hop in the chain, the latency to distribute Justin Bieber's babbling is typically just 500ms in a quad–tree.

  Rule of thumb: $bandwidth = latency$, i.e. if it takes 20ms from hop to hop, each node should replicate data for 20ms — if we make the tree wider, the linear effort of replicating data will dominate transfer time, if we make the tree more narrow, the hop–to–hop time will dominate.
- The tree–like graph greatly reduces the number of nodes to know

Possible Performance

- Trees with a bigger base reduce latency. Example: To transfer a Justin Bieber tweet to 50 million followers, a binary tree needs 25.5 hops on average, a quad–tree 12.8 hops, and an oct–tree 8.5 hops.
- A typical domestic (inside e.g. Germany) hop–to–hop time is just 20ms. International hops can be in the order of 250ms. Assuming there is only one international hop in the chain, the latency to distribute Justin Bieber's babbling is typically just 500ms in a quad–tree.
- Rule of thumb: *bandwidth* = *latency*, i.e. if it takes 20ms from hop to hop, each node should replicate data for 20ms — if we make the tree wider, the linear effort of replicating data will dominate transfer time, if we make the tree more narrow, the hop–to–hop time will dominate.
- The tree–like graph greatly reduces the number of nodes to know

Possible Performance

- Trees with a bigger base reduce latency. Example: To transfer a Justin Bieber tweet to 50 million followers, a binary tree needs 25.5 hops on average, a quad–tree 12.8 hops, and an oct–tree 8.5 hops.
- A typical domestic (inside e.g. Germany) hop–to–hop time is just 20ms. International hops can be in the order of 250ms. Assuming there is only one international hop in the chain, the latency to distribute Justin Bieber's babbling is typically just 500ms in a quad–tree.
- Rule of thumb: $bandwidth = latency$, i.e. if it takes 20ms from hop to hop, each node should replicate data for 20ms — if we make the tree wider, the linear effort of replicating data will dominate transfer time, if we make the tree more narrow, the hop–to–hop time will dominate.
- The tree–like graph greatly reduces the number of nodes to know

Distributed Prefix Hash Tree

- Most DHT approaches have poor performanc
- Prefix Hash Trees use a quite large base
- Only a few queries necessary to query an extremely large data base
- Suggestion: Active instantaneous replication of all changed data using the avalanche tree mentioned above

Distributed Prefix Hash Tree

- Most DHT approaches have poor performanc
- Prefix Hash Trees use a quite large base
- Only a few queries necessary to query an extremely large data base
- Suggestion: Active instantaneous replication of all changed data using the avalanche tree mentioned above

Distributed Prefix Hash Tree

- Most DHT approaches have poor performanc
- Prefix Hash Trees use a quite large base
- Only a few queries necessary to query an extremely large data base
- Suggestion: Active instantaneous replication of all changed data using the avalanche tree mentioned above

Distributed Prefix Hash Tree

- Most DHT approaches have poor performanc
- Prefix Hash Trees use a quite large base
- Only a few queries necessary to query an extremely large data base
- Suggestion: Active instantaneous replication of all changed data using the avalanche tree mentioned above

Content or Apps?

- The current web is defined by content — web apps (JavaScript) are an afterthough
- Therefore, the application logic is usually on the server side
- This doesn't work for a P2P network!

Content is structured text, images, videos, music, etc.

Content or Apps?

- The current web is defined by content — web apps (JavaScript) are an afterthough
- Therefore, the application logic is usually on the server side
- This doesn't work for a P2P network!

Content is structured text, images, videos, music, etc.

Content or Apps?

- The current web is defined by content — web apps (JavaScript) are an afterthough
- Therefore, the application logic is usually on the server side
- This doesn't work for a P2P network!

Content is structured text, images, videos, music, etc.

Content or Apps?

- The current web is defined by content — web apps (JavaScript) are an afterthough
- Therefore, the application logic is usually on the server side
- This doesn't work for a P2P network!
- Content is structured text, images, videos, music, etc.

App–Centric World

- There's a phenomenon I call "Turing creep": Every sufficiently complex system contains a user–accessible Turing–complete language
- Corollary: Every efficient sufficiently complex system can execute native machine code
- The application logic is to present the data; data itself is as above: structured text, images, videos, music, etc.
- Executing (especially efficient) code from the net raises obvious questions about security

App–Centric World

- There's a phenomenon I call "Turing creep": Every sufficiently complex system contains a user–accessible Turing–complete language
- Corollary: Every efficient sufficiently complex system can execute native machine code
- The application logic is to present the data; data itself is as above: structured text, images, videos, music, etc.
- Executing (especially efficient) code from the net raises obvious questions about security

App–Centric World

- There's a phenomenon I call "Turing creep": Every sufficiently complex system contains a user–accessible Turing–complete language
- Corollary: Every efficient sufficiently complex system can execute native machine code
- The application logic is to present the data; data itself is as above: structured text, images, videos, music, etc.
- Executing (especially efficient) code from the net raises obvious questions about security

App–Centric World

- There's a phenomenon I call "Turing creep": Every sufficiently complex system contains a user–accessible Turing–complete language
- Corollary: Every efficient sufficiently complex system can execute native machine code
- The application logic is to present the data; data itself is as above: structured text, images, videos, music, etc.
- Executing (especially efficient) code from the net raises obvious questions about security

How to securely execute code?

There are several options tried; as usual, things are broken:

1. Execute code in a controlled secure VM, see for example Java. This is broken by design, as securing something from the inside doesn't work.
2. Execute code in a sandbox. This has shown as more robust, depending on how complex the outside of the sandbox is.
3. Public inspection of code. This is how the open source world works, but the underhanded C contest shows that inspection is tricky.
4. Scan for known evil code. This is the security industry's approach, and it is not working.
5. Code signing can work together with public inspection — but using it for accountability doesn't work.

Therefore the choice is to sandbox public inspected code.

How to securely execute code?

There are several options tried; as usual, things are broken:

1. Execute code in a controlled secure VM, see for example Java. This is broken by design, as securing something from the inside doesn't work.
2. Execute code in a sandbox. This has shown as more robust, depending on how complex the outside of the sandbox is.
3. Public inspection of code. This is how the open source world works, but the underhanded C contest shows that inspection is tricky.
4. Scan for known evil code. This is the security industry's approach, and it is not working.
5. Code signing can work together with public inspection — but using it for accountability doesn't work.

Therefore the choice is to sandbox public inspected code.

How to securely execute code?

There are several options tried; as usual, things are broken:

1. Execute code in a controlled secure VM, see for example Java. This is broken by design, as securing something from the inside doesn't work.
2. Execute code in a sandbox. This has shown as more robust, depending on how complex the outside of the sandbox is.
3. Public inspection of code. This is how the open source world works, but the underhanded C contest shows that inspection is tricky.
4. Scan for known evil code. This is the security industry's approach, and it is not working.
5. Code signing can work together with public inspection — but using it for accountability doesn't work.

Therefore the choice is to sandbox public inspected code.

How to securely execute code?

There are several options tried; as usual, things are broken:

1. Execute code in a controlled secure VM, see for example Java. This is broken by design, as securing something from the inside doesn't work.
2. Execute code in a sandbox. This has shown as more robust, depending on how complex the outside of the sandbox is.
3. Public inspection of code. This is how the open source world works, but the underhanded C contest shows that inspection is tricky.
4. Scan for known evil code. This is the security industry's approach, and it is not working.
5. Code signing can work together with public inspection — but using it for accountability doesn't work.

Therefore the choice is to sandbox public inspected code.

How to securely execute code?

There are several options tried; as usual, things are broken:

1. Execute code in a controlled secure VM, see for example Java. This is broken by design, as securing something from the inside doesn't work.
2. Execute code in a sandbox. This has shown as more robust, depending on how complex the outside of the sandbox is.
3. Public inspection of code. This is how the open source world works, but the underhanded C contest shows that inspection is tricky.
4. Scan for known evil code. This is the security industry's approach, and it is not working.
5. Code signing can work together with public inspection — but using it for accountability doesn't work.

Therefore the choice is to sandbox public inspected code.

How to securely execute code?

There are several options tried; as usual, things are broken:

1. Execute code in a controlled secure VM, see for example Java. This is broken by design, as securing something from the inside doesn't work.
2. Execute code in a sandbox. This has shown as more robust, depending on how complex the outside of the sandbox is.
3. Public inspection of code. This is how the open source world works, but the underhanded C contest shows that inspection is tricky.
4. Scan for known evil code. This is the security industry's approach, and it is not working.
5. Code signing can work together with public inspection — but using it for accountability doesn't work

Therefore the choice is to sandbox public inspected code.

How to securely execute code?

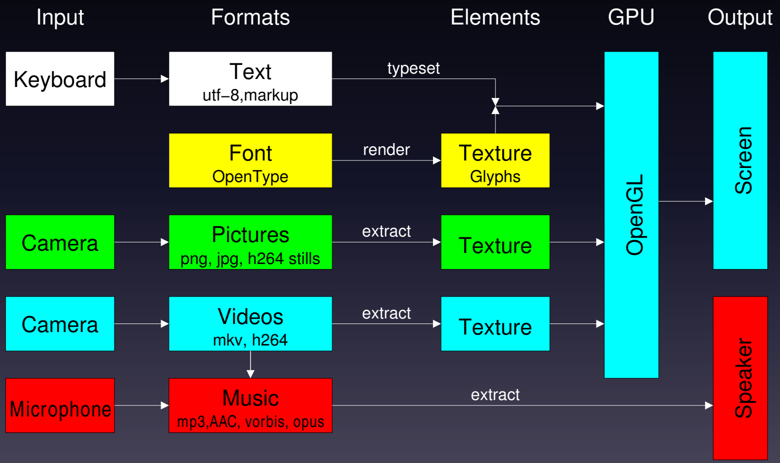
There are several options tried; as usual, things are broken:

1. Execute code in a controlled secure VM, see for example Java. This is broken by design, as securing something from the inside doesn't work.
2. Execute code in a sandbox. This has shown as more robust, depending on how complex the outside of the sandbox is.
3. Public inspection of code. This is how the open source world works, but the underhanded C contest shows that inspection is tricky.
4. Scan for known evil code. This is the security industry's approach, and it is not working.
5. Code signing can work together with public inspection — but using it for accountability doesn't work

Therefore the choice is to sandbox public inspected code.

Formats&Requirements

How to display things



Why OpenGL?

OpenGL can do everything

OpenGL renders:

1. Triangles, lines, points — simple components
2. Textures and gradients
3. and uses shader programs — the most powerful thing in OpenGL from 2.0.

Real requirement: visualization of *any* data. OpenGL can do that.

Why OpenGL?

OpenGL can do everything

OpenGL renders:

1. Triangles, lines, points — simple components
2. Textures and gradients
3. and uses shader programs — the most powerful thing in OpenGL from 2.0.

Real requirement: visualization of *any* data. OpenGL can do that.

Why OpenGL?

OpenGL can do everything

OpenGL renders:

1. Triangles, lines, points — simple components
2. Textures and gradients
3. and uses shader programs — the most powerful thing in OpenGL from 2.0.

Real requirement: visualization of *any* data. OpenGL can do that.

Why OpenGL?

OpenGL can do everything

OpenGL renders:

1. Triangles, lines, points — simple components
2. Textures and gradients
3. and uses shader programs — the most powerful thing in OpenGL from 2.0.

Real requirement: visualization of *any* data. OpenGL can do that.

Why OpenGL?

OpenGL can do everything

OpenGL renders:

1. Triangles, lines, points — simple components
2. Textures and gradients
3. and uses shader programs — the most powerful thing in OpenGL from 2.0.

Real requirement: visualization of *any* data. OpenGL can do that.

How to connect the media?

Lemma: every glue logic will become Turing complete

- currently used glue: HTML+CSS+JavaScript
- containers with Flash, Java, ActiveX, PDF, Google's NaCl...
- conclusion: use a powerful tool right from start!

browser: run–time and development tool for applications

How to connect the media?

Lemma: every glue logic will become Turing complete

- currently used glue: HTML+CSS+JavaScript
- containers with Flash, Java, ActiveX, PDF, Google's NaCl...
- conclusion: use a powerful tool right from start!

  browser: run–time and development tool for applications

How to connect the media?

Lemma: every glue logic will become Turing complete

- currently used glue: HTML+CSS+JavaScript
- containers with Flash, Java, ActiveX, PDF, Google's NaCl...
- conclusion: use a powerful tool right from start!

browser: run–time and development tool for applications

How to connect the media?

Lemma: every glue logic will become Turing complete

- currently used glue: HTML+CSS+JavaScript
- containers with Flash, Java, ActiveX, PDF, Google's NaCl...
- conclusion: use a powerful tool right from start!
- browser: run–time and development tool for applications

Frameworks

- libsoil for images (PNG+JPEG loading into a texture)
- freetype-gl for fonts (TrueType/OpenType into a texture)
- OpenMAX on Android, gstreamer on Linux: videos into a texture
- MINOΣ2: Lightweight OpenGL–based widget library in Forth (still a lot of work in progress)

Frameworks

- libsoil for images (PNG+JPEG loading into a texture)
- freetype-gl for fonts (TrueType/OpenType into a texture)
- OpenMAX on Android, gstreamer on Linux: videos into a texture
- MINOΣ2: Lightweight OpenGL–based widget library in Forth (still a lot of work in progress)

Frameworks

- libsoil for images (PNG+JPEG loading into a texture)
- freetype-gl for fonts (TrueType/OpenType into a texture)
- OpenMAX on Android, gstreamer on Linux: videos into a texture
- MINOΣ2: Lightweight OpenGL–based widget library in Forth (still a lot of work in progress)

Frameworks

- libsoil for images (PNG+JPEG loading into a texture)
- freetype-gl for fonts (TrueType/OpenType into a texture)
- OpenMAX on Android, gstreamer on Linux: videos into a texture
- MINOΣ2: Lightweight OpenGL–based widget library in Forth (still a lot of work in progress)

For Further Reading I

Bernd Paysan
net2o source repository and wiki
http://fossil.net2o.de/net2o