net2o: Transport Layer — Implemented

Tame the Net

Bernd Paysan

EuroForth 2012, Oxford

Overview

Motivation

Datenflusssteuerung

Reliability

Cryptography

Recap: What’s Broken?

- TCP–Flow Control: “Buffer Bloat”
- TCP as “carefree protocol” is not even remotely real–time capable, so far from “carefree” for media use
- UDP is only a “easy” access to raw IP, and otherwise “do it yourself”
- The SSL–PKI with their “honest Achmeds” as certification authorities
- Encryption “too complicated, too difficult”, usually added late, and therefore way too often not done

Recap: What’s Broken?

- TCP–Flow Control: “Buffer Bloat”
- TCP as “carefree protocol” is not even remotely real–time capable, so far from “carefree” for media use
- UDP is only a “easy” access to raw IP, and otherwise “do it yourself”
- The SSL–PKI with their “honest Achmeds” as certification authorities
- Encryption “too complicated, too difficult”, usually added late, and therefore way too often not done

Recap: What’s Broken?

- TCP–Flow Control: “Buffer Bloat”
- TCP as “carefree protocol” is not even remotely real–time capable, so far from “carefree” for media use
- UDP is only a “easy” access to raw IP, and otherwise “do it yourself”
- The SSL–PKI with their “honest Achmeds” as certification authorities
- Encryption “too complicated, too difficult”, usually added late, and therefore way too often not done

Recap: What’s Broken?

- TCP–Flow Control: “Buffer Bloat”
- TCP as “carefree protocol” is not even remotely real–time capable, so far from “carefree” for media use
- UDP is only a “easy” access to raw IP, and otherwise “do it yourself”
- The SSL–PKI with their “honest Achmeds” as certification authorities
- Encryption “too complicated, too difficult”, usually added late, and therefore way too often not done

Recap: What’s Broken?

- TCP–Flow Control: “Buffer Bloat”
- TCP as “carefree protocol” is not even remotely real–time capable, so far from “carefree” for media use
- UDP is only a “easy” access to raw IP, and otherwise “do it yourself”
- The SSL–PKI with their “honest Achmeds” as certification authorities
- Encryption “too complicated, too difficult”, usually added late, and therefore way too often not done

Changes from the Draft

- Packet size now $64 * 2^n, \quad n \in \{0, \ldots, 15\}$, so up to 2MB in powers of 2
- No “embedded” variant implemented, only 64 bit addresses
- Routing address length changed to 128 bits
- Encryption always active
- No “salt” at the start of a packet, but a cryptographic checksum (128 bit) at the end

Changes from the Draft

- Packet size now $64 * 2^n, \quad n \in \{0,\ldots,15\}$, so up to 2MB in powers of 2
- No “embedded” variant implemented, only 64 bit addresses
- Routing address length changed to 128 bits
- Encryption always active
- No “salt” at the start of a packet, but a cryptographic checksum (128 bit) at the end

Changes from the Draft

- Packet size now $64 * 2^n, \quad n \in \{0, \ldots, 15\}$, so up to 2MB in powers of 2
- No “embedded” variant implemented, only 64 bit addresses
- Routing address length changed to 128 bits
- Encryption always active
- No “salt” at the start of a packet, but a cryptographic checksum (128 bit) at the end

Changes from the Draft

- Packet size now $64 * 2^n, \quad n \in \{0, \dots, 15\}$, so up to 2MB in powers of 2
- No “embedded” variant implemented, only 64 bit addresses
- Routing address length changed to 128 bits
- Encryption always active
- No “salt” at the start of a packet, but a cryptographic checksum (128 bit) at the end

Changes from the Draft

- Packet size now $64 * 2^n, \quad n \in \{0, \ldots, 15\}$, so up to 2MB in powers of 2
- No “embedded” variant implemented, only 64 bit addresses
- Routing address length changed to 128 bits
- Encryption always active
- No “salt” at the start of a packet, but a cryptographic checksum (128 bit) at the end

Status: TCP Flow Control

- 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.

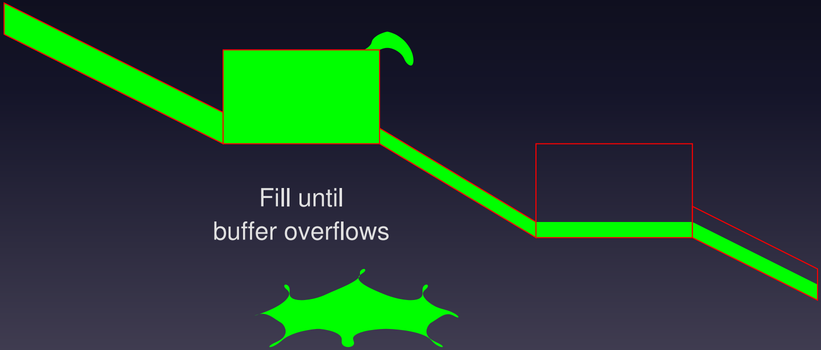

Figure: Buffer Bloat

Alternatives?

- LEDBAT tries to achieve a low, constant delay: Works, but not good on fairness
- CurveCP has a similar approach, which is not even documented (but DAN BERNSTEIN's code is by definition "obvious")
- Therefore, something new has to be done

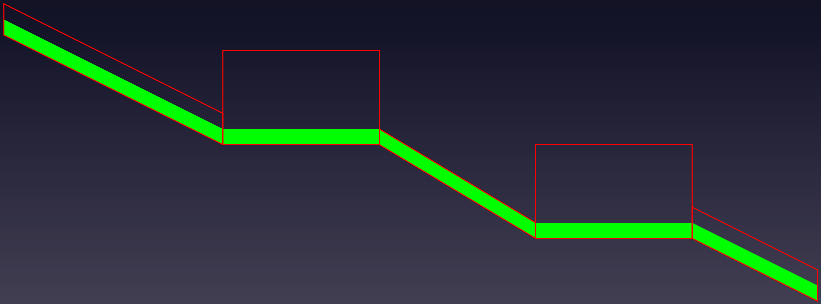

Figure: That's how proper flow control should look like

Alternatives?

- LEDBAT tries to achieve a low, constant delay: Works, but not good on fairness
- CurveCP has a similar approach, which is not even documented (but DAN BERNSTEIN's code is by definition "obvious")
- Therefore, something new has to be done

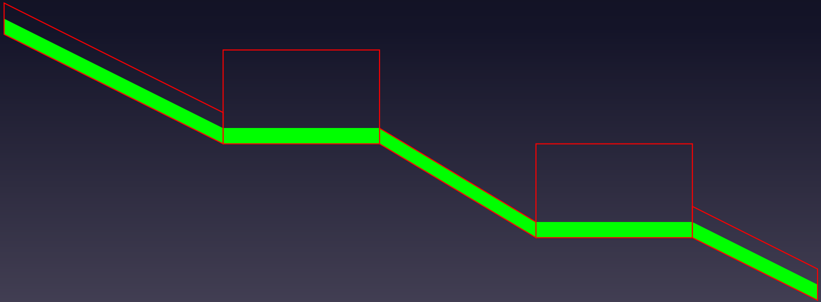

Figure: That's how proper flow control should look like

Alternatives?

- LEDBAT tries to achieve a low, constant delay: Works, but not good on fairness
- CurveCP has a similar approach, which is not even documented (but DAN BERNSTEIN's code is by definition "obvious")
- Therefore, something new has to be done

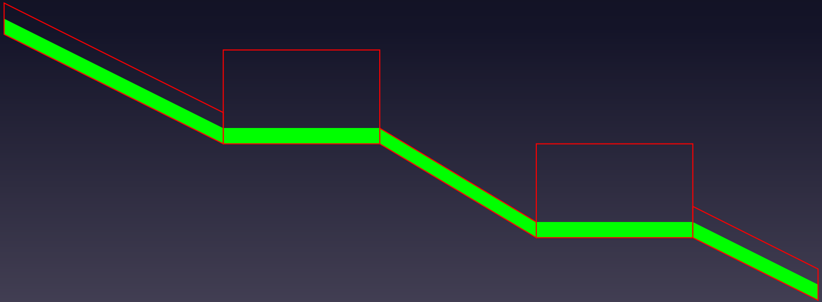

Figure: That's how proper flow control should look like

„Buffer Bloat“

- Retransmits are making the situation worse in case of congestions and therefore should be avoided
- Riddle: How big should the buffer be, under the assumption that the bandwidth is used optimally,the bottleneck is on the other side of the connection, and a second data stream is opened up?
- Answer: about half the round trip delay, which are inevitably filled before any reaction is possible
- Buffers are good, but you shouldn’t fill them up to the brim
- The problem is inherent in the TCP protocol, but since Windows XP did not provide window scaling, the per-connection buffer limit was 64k for most connections on the Internet for quite a long time.

„Buffer Bloat“

- Retransmits are making the situation worse in case of congestions and therefore should be avoided
- Riddle: How big should the buffer be, under the assumption that the bandwidth is used optimally,the bottleneck is on the other side of the connection, and a second data stream is opened up?
- Answer: about half the round trip delay, which are inevitably filled before any reaction is possible
- Buffers are good, but you shouldn’t fill them up to the brim
- The problem is inherent in the TCP protocol, but since Windows XP did not provide window scaling, the per-connection buffer limit was OK for most connections on the Internet for quite a long time.

„Buffer Bloat“

- Retransmits are making the situation worse in case of congestions and therefore should be avoided
- Riddle: How big should the buffer be, under the assumption that the bandwidth is used optimally,the bottleneck is on the other side of the connection, and a second data stream is opened up?
- Answer: about half the round trip delay, which are inevitably filled before any reaction is possible
- Buffers are good, but you shouldn’t fill them up to the brim
- The problem is inherent in the TCP protocol, but since Windows XP did not provide window scaling, the per-connection buffer limit was OK for most connections on the Internet for quite a long time.

„Buffer Bloat“

- Retransmits are making the situation worse in case of congestions and therefore should be avoided
- Riddle: How big should the buffer be, under the assumption that the bandwidth is used optimally,the bottleneck is on the other side of the connection, and a second data stream is opened up?
- Answer: about half the round trip delay, which are inevitably filled before any reaction is possible
- Buffers are good, but you shouldn’t fill them up to the brim
- The problem is inherent in the TCP protocol, but since Windows XP did not provide window scaling, the per-connection buffer limit was 64k for most connections on the Internet for quite a long time.

„Buffer Bloat“

- Retransmits are making the situation worse in case of congestions and therefore should be avoided
- Riddle: How big should the buffer be, under the assumption that the bandwidth is used optimally,the bottleneck is on the other side of the connection, and a second data stream is opened up?
- Answer: about half the round trip delay, which are inevitably filled before any reaction is possible
- Buffers are good, but you shouldn’t fill them up to the brim
- The problem is inherent in the TCP protocol, but since Windows XP did not provide window scaling, the per–connection buffer limit was 64k for most connections on the Internet for quite a long time.

net2o Flow Control

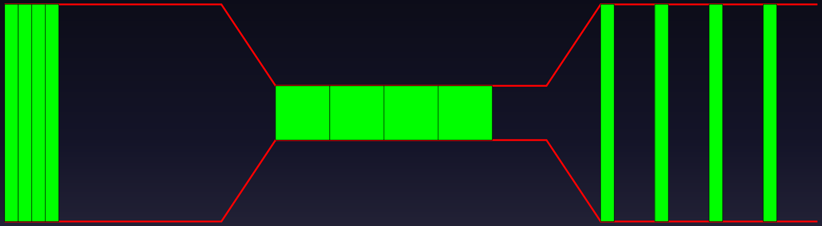

Figure: Measure the bottleneck using a burst of packets

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*.

```
: calc-rate ( -- )
  delta-ticks @ tick-init 1+ acks @ */
  lit, set-rate ;
```

Server would simply use this rate

```
: set-rate ( rate -- ) ns/burst ! ;
```

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*.

```
: calc-rate ( -- )
  delta-ticks @ tick-init 1+ acks @ */
  lit, set-rate ;
```

Server would simply use this rate

```
: set-rate ( rate -- )  ns/burst ! ;
```

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 relieve 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 relieve 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 relieve 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 relieve 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

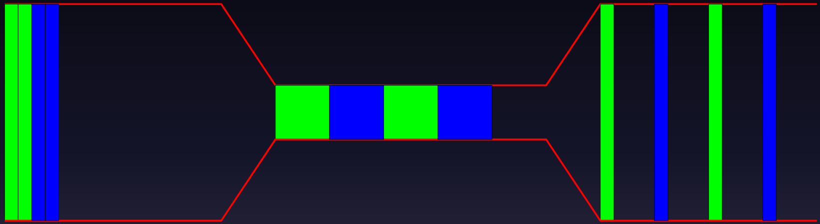

Figure: Fair queuing results in correct measurement of available bandwidth

net2o Flow Control — FIFO Router

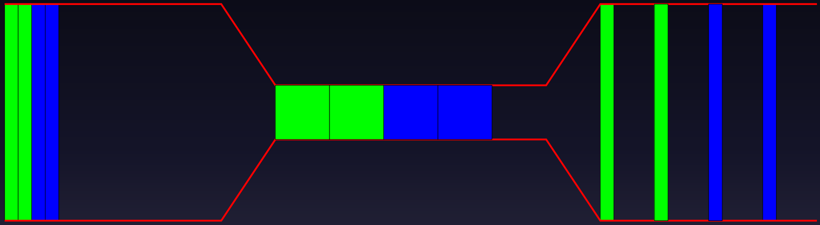

Figure: Unfair FIFO queuing results in twice the available bandwidth calculated

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
- $\max(slacks)/10ms$ 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.

VDSL

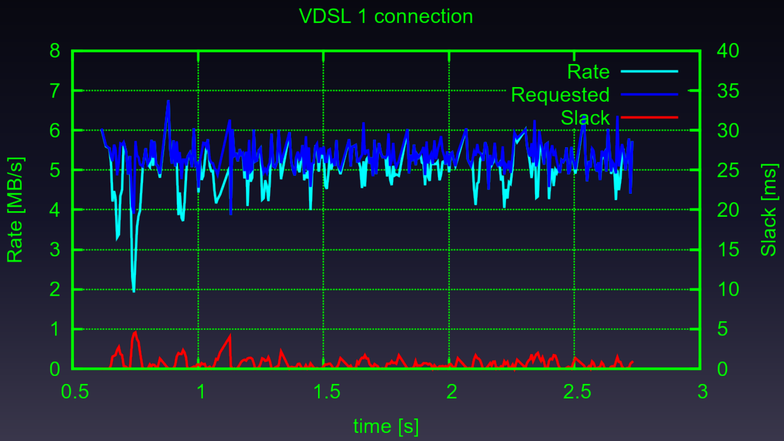

Figure: One connection on a VDSL

VDSL, Congestion

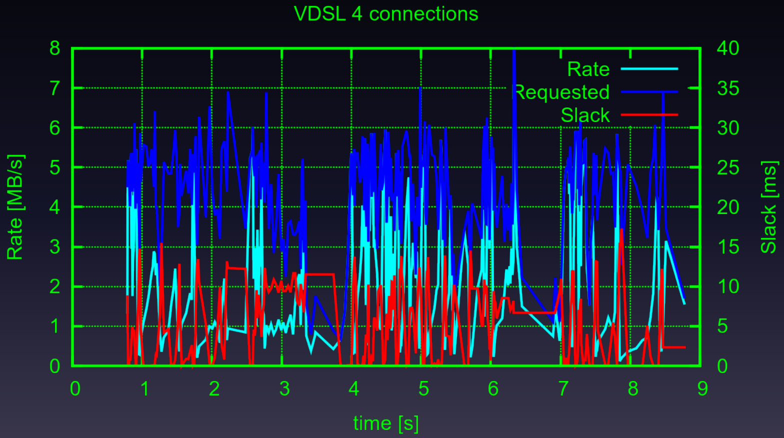

Figure: One of four connections on a VDSL

Unreliable Air Cable (WLAN)

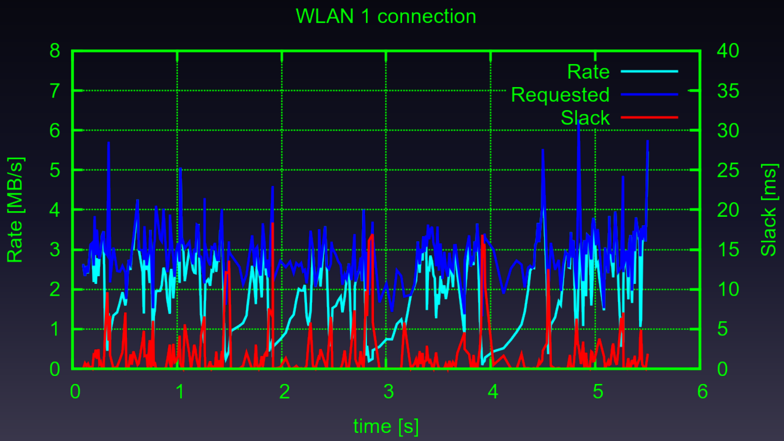

Figure: Single connection using WLAN

Unreliable Air Cable, Congestion

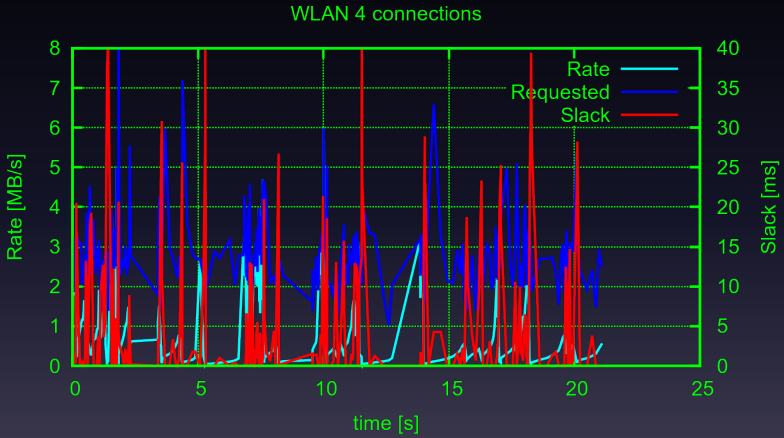

Figure: One of four connections using WLAN

Transport Reliability

- Packet ordering is dealt with the address each packet carries
- The receiver tracks received packets in two alternating bitmaps
- Received packets are marked as received in the active bitmap
- The other bitmap is filled up, until the bitmaps are swapped (twice per round trip delay RTD)
- Wait one RTD for retransmits
- Retransmits are preferred, but no timing measurement on retransmits (two identical packets in flight)

Transport Reliability

- Packet ordering is dealt with the address each packet carries
- The receiver tracks received packets in two alternating bitmaps
- Received packets are marked as received in the active bitmap
- The other bitmap is filled up, until the bitmaps are swapped (twice per round trip delay RTD)
- Wait one RTD for retransmits
- Retransmits are preferred, but no timing measurement on retransmits (two identical packets in flight)

Transport Reliability

- Packet ordering is dealt with the address each packet carries
- The receiver tracks received packets in two alternating bitmaps
- Received packets are marked as received in the active bitmap
- The other bitmap is filled up, until the bitmaps are swapped (twice per round trip delay RTD)
- Wait one RTD for retransmits
- Retransmits are preferred, but no timing measurement on retransmits (two identical packets in flight)

Transport Reliability

- Packet ordering is dealt with the address each packet carries
- The receiver tracks received packets in two alternating bitmaps
- Received packets are marked as received in the active bitmap
- The other bitmap is filled up, until the bitmaps are swapped (twice per round trip delay RTD)
- Wait one RTD for retransmits
- Retransmits are preferred, but no timing measurement on retransmits (two identical packets in flight)

Transport Reliability

- Packet ordering is dealt with the address each packet carries
- The receiver tracks received packets in two alternating bitmaps
- Received packets are marked as received in the active bitmap
- The other bitmap is filled up, until the bitmaps are swapped (twice per round trip delay RTD)
- Wait one RTD for retransmits
- Retransmits are preferred, but no timing measurement on retransmits (two identical packets in flight)

Transport Reliability

- Packet ordering is dealt with the address each packet carries
- The receiver tracks received packets in two alternating bitmaps
- Received packets are marked as received in the active bitmap
- The other bitmap is filled up, until the bitmaps are swapped (twice per round trip delay RTD)
- Wait one RTD for retransmits
- Retransmits are preferred, but no timing measurement on retransmits (two identical packets in flight)

Reliable Execution of Commands

- The command block at that address is received first time ⟶ execute, remember the reply command
- The command block has already been received ⟶ send the reply again (don't execute the command)
- No replies requested ⟶ Do nothing
- Acknowledges are amended by a checksum, which only the sender or the receiver can compute, so no fake acknowledge for dropped packets is possible.

Reliable Execution of Commands

- The command block at that address is received first time ⟶ execute, remember the reply command
- The command block has already been received ⟶ send the reply again (don't execute the command)
- No replies requested ⟶ Do nothing
- Acknowledges are amended by a checksum, which only the sender or the receiver can compute, so no fake acknowledge for dropped packets is possible.

Reliable Execution of Commands

- The command block at that address is received first time ⟶ execute, remember the reply command
- The command block has already been received ⟶ send the reply again (don’t execute the command)
- No replies requested ⟶ Do nothing
- Acknowledges are amended by a checksum, which only the sender or the receiver can compute, so no fake acknowledge for dropped packets is possible.

Reliable Execution of Commands

- The command block at that address is received first time ⟶ execute, remember the reply command
- The command block has already been received ⟶ send the reply again (don’t execute the command)
- No replies requested ⟶ Do nothing
- Acknowledges are amended by a checksum, which only the sender or the receiver can compute, so no fake acknowledge for dropped packets is possible.

Cryptography

Communication needs the first three goals, the fourth one isn’t

Confidentiality no third party (Eve) should eavesdrop the communication

Integrity The data is complete and unmodified

Authentication The sender of the data can be identified

Non–repudiation is not necessary for two–way communication

Cryptography

Communication needs the first three goals, the fourth one isn't

Confidentiality no third party (Eve) should eavesdrop the communication

Integrity The data is complete and unmodified

Authentication The sender of the data can be identified

Non–repudiation is not necessary for two–way communication

Cryptography

Communication needs the first three goals, the fourth one isn’t

Confidentiality no third party (Eve) should eavesdrop the communication

Integrity The data is complete and unmodified

Authentication The sender of the data can be identified

Non–repudiation is not necessary for two–way communication

Cryptography

Communication needs the first three goals, the fourth one isn’t

Confidentiality no third party (Eve) should eavesdrop the communication

Integrity The data is complete and unmodified

Authentication The sender of the data can be identified

Non–repudiation is not necessary for two–way communication

Cryptography

Communication needs the first three goals, the fourth one isn’t

Confidentiality no third party (Eve) should eavesdrop the communication

Integrity The data is complete and unmodified

Authentication The sender of the data can be identified

Non–repudiation is not necessary for two–way communication

Used Technology: Curve25519

- Elliptic Curve Cryptography doesn’t base on large number factoring (as hard to solve problems), but on natural logarithms of elliptic curves
- Security level of Curve25519 corresponds to 128 bits in a symmetric key — that’s sufficient today
- Curve25519 has a very efficient implementation
- It is optimized for 1:1 connections
- Each participant “multiplies” his secret key with the public key of the other side, both products are identical

Used Technology: Curve25519

- Elliptic Curve Cryptography doesn’t base on large number factoring (as hard to solve problems), but on natural logarithms of elliptic curves
- Security level of Curve25519 corresponds to 128 bits in a symmetric key — that’s sufficient today
- Curve25519 has a very efficient implementation
- It is optimized for 1:1 connections
- Each participant “multiplies” his secret key with the public key of the other side, both products are identical

Used Technology: Curve25519

- Elliptic Curve Cryptography doesn’t base on large number factoring (as hard to solve problems), but on natural logarithms of elliptic curves
- Security level of Curve25519 corresponds to 128 bits in a symmetric key — that’s sufficient today
- Curve25519 has a very efficient implementation
- It is optimized for 1:1 connections
- Each participant “multiplies” his secret key with the public key of the other side, both products are identical

Used Technology: Curve25519

- Elliptic Curve Cryptography doesn’t base on large number factoring (as hard to solve problems), but on natural logarithms of elliptic curves
- Security level of Curve25519 corresponds to 128 bits in a symmetric key — that’s sufficient today
- Curve25519 has a very efficient implementation
- It is optimized for 1:1 connections
- Each participant “multiplies” his secret key with the public key of the other side, both products are identical

Used Technology: Curve25519

- Elliptic Curve Cryptography doesn’t base on large number factoring (as hard to solve problems), but on natural logarithms of elliptic curves
- Security level of Curve25519 corresponds to 128 bits in a symmetric key — that’s sufficient today
- Curve25519 has a very efficient implementation
- It is optimized for 1:1 connections
- Each participant “multiplies” his secret key with the public key of the other side, both products are identical

Wurstkessel

At the moment, I’m using Wurstkessel as symmetric encryption, even though there hasn’t been a thorough review:

- Wurstkessel provides en/decryption and authentication in a single pass, computing a key–dependent secure hash
- Thus a single run of Wurstkessel solves all three tasks: confidentiality, integrity, and authentication.
  1. the data is encrypted
  2. the correct hash proves its integrity
  3. the hash can only be calculated knowing the key, therefore proving the authentication of the sender
- AES has something similar, the CBC–MAC. However, in AES, it is necessary to use different keys for encryption and MAC, i.e. no single run possible

Wurstkessel

At the moment, I’m using Wurstkessel as symmetric encryption, even though there hasn’t been a thorough review:

- Wurstkessel provides en/decryption and authentication in a single pass, computing a key–dependent secure hash
- Thus a single run of Wurstkessel solves all three tasks: confidentiality, integrity, and authentication.
  1. the data is encrypted
  2. the correct hash proves its integrity
  3. the hash can only be calculated knowing the key, therefore proving the authentication of the sender
- AES has something similar, the CBC–MAC. However, in AES, it is necessary to use different keys for encryption and MAC, i.e. no single run possible

Wurstkessel

At the moment, I'm using Wurstkessel as symmetric encryption, even though there hasn't been a thorough review:

- Wurstkessel provides en/decryption and authentication in a single pass, computing a key–dependent secure hash
- Thus a single run of Wurstkessel solves all three tasks: confidentiality, integrity, and authentication.
  1. the data is encrypted
  2. the correct hash proves its integrity
  3. the hash can only be calculated knowing the key, therefore proving the authentication of the sender
- AES has something similar, the CBC–MAC. However, in AES, it is necessary to use different keys for encryption and MAC, i.e. no single run possible

Wurstkessel

At the moment, I'm using Wurstkessel as symmetric encryption, even though there hasn't been a thorough review:

- Wurstkessel provides en/decryption and authentication in a single pass, computing a key–dependent secure hash
- Thus a single run of Wurstkessel solves all three tasks: confidentiality, integrity, and authentication.
  1. the data is encrypted
  2. the correct hash proves its integrity
  3. the hash can only be calculated knowing the key, therefore proving the authentication of the sender
- AES has something similar, the CBC–MAC. However, in AES, it is necessary to use different keys for encryption and MAC, i.e. no single run possible

Wurstkessel

At the moment, I’m using Wurstkessel as symmetric encryption, even though there hasn’t been a thorough review:

- Wurstkessel provides en/decryption and authentication in a single pass, computing a key–dependent secure hash
- Thus a single run of Wurstkessel solves all three tasks: confidentiality, integrity, and authentication.
  1. the data is encrypted
  2. the correct hash proves its integrity
  3. the hash can only be calculated knowing the key, therefore proving the authentication of the sender
- AES has something similar, the CBC–MAC. However, in AES, it is necessary to use different keys for encryption and MAC, i.e. no single run possible

Wurstkessel

At the moment, I'm using Wurstkessel as symmetric encryption, even though there hasn't been a thorough review:

- Wurstkessel provides en/decryption and authentication in a single pass, computing a key–dependent secure hash
- Thus a single run of Wurstkessel solves all three tasks: confidentiality, integrity, and authentication.
    1. the data is encrypted
    2. the correct hash proves its integrity
    3. the hash can only be calculated knowing the key, therefore proving the authentication of the sender
- AES has something similar, the CBC–MAC. However, in AES, it is necessary to use different keys for encryption and MAC, i.e. no single run possible

Wurstkessel

At the moment, I'm using Wurstkessel as symmetric encryption, even though there hasn't been a thorough review:

- Wurstkessel provides en/decryption and authentication in a single pass, computing a key–dependent secure hash
- Thus a single run of Wurstkessel solves all three tasks: confidentiality, integrity, and authentication.
    1. the data is encrypted
    2. the correct hash proves its integrity
    3. the hash can only be calculated knowing the key, therefore proving the authentication of the sender
- AES has something similar, the CBC–MAC. However, in AES, it is necessary to use different keys for encryption and MAC, i.e. no single run possible

Hidden Initialization Vectors

- No key reuse allowed (only for retransmissions), otherwise a known–plaintext attack is possible
- Usual approach: initialization vector (IV) transmitted with each packet
- Disadvantage: Overhead and the “other” part of the key is known to the attacker
- Solution: Generate the IVs using a PRNG (with Wurstkessel in PRNG mode) on both sides — these IVs are “shared secrets”. Only the seed for the PRNG is transmitted, and used together with the shared key to generate the IVs (Idea: Helmar Wodke)

Hidden Initialization Vectors

- No key reuse allowed (only for retransmissions), otherwise a known–plaintext attack is possible
- Usual approach: initialization vector (IV) transmitted with each packet
- Disadvantage: Overhead and the “other” part of the key is known to the attacker
- Solution: Generate the IVs using a PRNG (with Wurstkessel in PRNG mode) on both sides — these IVs are “shared secrets”. Only the seed for the PRNG is transmitted, and used together with the shared key to generate the IVs (Idea: Helmar Wodke)

Hidden Initialization Vectors

- No key reuse allowed (only for retransmissions), otherwise a known–plaintext attack is possible
- Usual approach: initialization vector (IV) transmitted with each packet
- Disadvantage: Overhead and the “other” part of the key is known to the attacker
- Solution: Generate the IVs using a PRNG (with Wurstkessel in PRNG mode) on both sides — these IVs are “shared secrets”. Only the seed for the PRNG is transmitted, and used together with the shared key to generate the IVs (Idea: HELMAR WODKE).

Hidden Initialization Vectors

- No key reuse allowed (only for retransmissions), otherwise a known–plaintext attack is possible
- Usual approach: initialization vector (IV) transmitted with each packet
- Disadvantage: Overhead and the “other” part of the key is known to the attacker
- Solution: Generate the IVs using a PRNG (with Wurstkessel in PRNG mode) on both sides — these IVs are “shared secrets”. Only the seed for the PRNG is transmitted, and used together with the shared key to generate the IVs (Idea: Helmar Wodke).

Public Key Infrastructure (PKI)

At the moment, three approaches are used:

1. Hierarchical Certification Authorities (e.g. SSL): The trust is delegated to “notaries”, i.e. the CAs, which then must be trustworthy (all of them, since each CA can create a certificate for anybody). The server is certified, i.e. the user knows that he can trust this connection as much as the worst of those 600 CAs.
2. Peer to Peer (e.g. PGP): trust is obtained through a “web of trust”, i.e. you either trust directly or by using several people you trust. It is not sufficient to corrupt a single person in your trust network to obtain trust.
3. Observing changes (e.g. SSH): trust is reiterated by repeated contacts, and as long as keys don’t change, trust is assumed.

Public Key Infrastructure (PKI)

At the moment, three approaches are used:

1. Hierarchical Certification Authorities (e.g. SSL): The trust is delegated to “notaries”, i.e. the CAs, which then must be trustworthy (all of them, since each CA can create a certificate for anybody). The server is certified, i.e. the user knows that he can trust this connection as much as the worst of those 600 CAs.
2. Peer to Peer (e.g. PGP): trust is obtained through a “web of trust”, i.e. you either trust directly or by using several people you trust. It is not sufficient to corrupt a single person in your trust network to obtain trust.
3. Observing changes (e.g. SSH): trust is reiterated by repeated contacts, and as long as keys don’t change, trust is assumed.

Public Key Infrastructure (PKI)

At the moment, three approaches are used:

1. Hierarchical Certification Authorities (e.g. SSL): The trust is delegated to “notaries”, i.e. the CAs, which then must be trustworthy (all of them, since each CA can create a certificate for anybody). The server is certified, i.e. the user knows that he can trust this connection as much as the worst of those 600 CAs.
2. Peer to Peer (e.g. PGP): trust is obtained through a “web of trust”, i.e. you either trust directly or by using several people you trust. It is not sufficient to corrupt a single person in your trust network to obtain trust.
3. Observing changes (e.g. SSH): trust is reiterated by repeated contacts, and as long as keys don't change, trust is assumed.

Public Key Infrastructure (PKI)

At the moment, three approaches are used:

1. Hierarchical Certification Authorities (e.g. SSL): The trust is delegated to “notaries”, i.e. the CAs, which then must be trustworthy (all of them, since each CA can create a certificate for anybody). The server is certified, i.e. the user knows that he can trust this connection as much as the worst of those 600 CAs.
2. Peer to Peer (e.g. PGP): trust is obtained through a “web of trust”, i.e. you either trust directly or by using several people you trust. It is not sufficient to corrupt a single person in your trust network to obtain trust.
3. Observing changes (e.g. SSH): trust is reiterated by repeated contacts, and as long as keys don’t change, trust is assumed.

What Was the Problem?

The typical reason to use a trusted connection is to obtain a secure login, and then access private data. This begs a question:

- Isn’t it actually the *client,* which should be trusted?

The connection is a trusted connection, if *one* participant has successfully evaluated the trust of the other.
Therefore, by inverting the trust relation, the SSH approach is sufficient in most cases.

What Was the Problem?

The typical reason to use a trusted connection is to obtain a secure login, and then access private data. This begs a question:

- Isn’t it actually the *client,* which should be trusted?

The connection is a trusted connection, if *one* participant has successfully evaluated the trust of the other.
Therefore, by inverting the trust relation, the SSH approach is sufficient in most cases.

What Was the Problem?

The typical reason to use a trusted connection is to obtain a secure login, and then access private data. This begs a question:

- Isn’t it actually the *client,* which should be trusted?

The connection is a trusted connection, if *one* participant has successfully evaluated the trust of the other.

Therefore, by inverting the trust relation, the SSH approach is sufficient in most cases.

What Was the Problem?

The typical reason to use a trusted connection is to obtain a secure login, and then access private data. This begs a question:

- Isn’t it actually the *client,* which should be trusted?

The connection is a trusted connection, if *one* participant has successfully evaluated the trust of the other.
Therefore, by inverting the trust relation, the SSH approach is sufficient in most cases.

For Further Reading

BERND PAYSAN
Fossil Repository und Wiki
`http://fossil.net2o.de/`