Network Protocol Design and Evaluation

06 - Design Techniques

Stefan Rührup

University of Freiburg
Computer Networks and Telematics
Summer 2009
Overview

‣ In the last lectures:
  • Specification and Verification

‣ In this part:
  • Design and implementation techniques
Design Decisions

- Communication protocols are subject to resource constraints.
- A communication protocol can be part of a larger system (protocol stack, application), which adds additional constraints.
- Resource constraints might be dependent or conflicting, and meeting all constraints is not always possible.
- Design techniques help to find trade-offs.
Resource constraints

- System design is constrained by resource limitations.

- **Basic resource constraints:**
  - Time (response time, throughput)
  - Space (memory, buffer capacity, bandwidth)
  - Computation
  - Labor
  - Money

- **Social constraints**
  - Standards
  - Market requirements

[Keshav 1997]
Bottlenecks

- Identify the most constrained resource, the binding constraint or bottleneck
- Removing this bottleneck can open other bottlenecks
- Goal: Balancing the whole system
- This is often infeasible. However, there are some design techniques to find trade-offs
- Methodology: Start with identifying constraints, then trade-off one resource for another to maximize utility.

[Keshav 1997]
Multiplexing

- Resource sharing
- Trade-off: Time and space vs. money
- Examples:
  - One server processes client requests simultaneously instead of setting up more servers.
  - If the communication medium is the bottleneck, it can be divided by frequency, or time slots to allow simultaneous communication between different communication partners.

[Keshav 1997]
Parallelism (1)

- Splitting tasks into independent subtasks
- Trade-off: computation vs. time
- Examples:
  - Web browsers download linked images from webpages in parallel.
  - Layers of a protocol stack can process their packets in parallel.

[Keshav 1997]
Parallelism (2)

- **Degree of Parallelism**
  \[
  \text{throughput} \times \text{response time} = \text{degree of parallelism}
  \]

- Response time = mean time to complete a task [sec/task]
- Throughput = mean number of tasks that can be completed within a unit of time [tasks/sec].

**Example:** Packet processing

![Diagram of network protocol](image)

- Slowest stage takes time T
- Throughput = \(1/T\)
- Degree of Parallelism = \(R/T\)

[Keshav 1997]
Batching (1)

- Group tasks together to level the overhead
- Trade-off: Response time vs. throughput
- Example: A remote login application accumulates typed characters and sends them in a batch instead of transmitting each character in a separate packet.
- Batching is only efficient, if the overhead for \( N \) tasks is smaller than \( N \) times the overhead for a single task

[Keshav 1997]
Batching (2)

- Worst-case response time for a task: $T + O$
- Worst-case throughput: $1/(T+O)$
- Assume the overhead $O'$ for a batch of $N$ tasks is smaller than $N \times O$.
- Worst-case response time for the batch: $A + N \times T + O'$
  where $A$ is the time for accumulating the tasks
- Worst-case throughput: $N/(N \times T + O') = 1/(T + O'/N)$
  (if we assume that the system can process other tasks during accumulation)

[Keshav 1997]
Locality

- Exploiting locality means that data that was accessed often will be kept in fast memory (caching).
- Trade-off: Space vs. time
- Example: During file transfer the sender splits a file into several packets. If it keeps the unacknowledged packets in fast memory, it can retransmit them without generating them again.

[Keshav 1997]
Hierarchy

- Decomposition of a system into smaller subsystems
- Increases scalability
- Example:
  - Hierarchical Addressing (IP Addresses)
  - Internet Domain Name System

DNS: The name space is partitioned into domains. Each domain has an associated DNS server.

[Keshav 1997]
Binding and Indirection

- Binding: Referring from an abstraction to an instance
- Indirection: Using the abstraction and dereferencing it automatically
- Examples:
  - eMail aliases
  - In a cellular telephone system, a user may move from one cell to another, but remains reachable by the same number. The system binds the user to a particular cell while the switches use indirection.

[Keshav 1997]
Virtualization

- Combination of multiplexing and indirection
- Allows sharing a resource as if it could be used exclusively
- Examples:
  - Virtual Private Network
  - Virtual Modem

[Keshav 1997]
Randomization

- Powerful tool to increase robustness
- Examples:
  - Ethernet channel access: After a packet collision, a jam signal is sent to make sure that all participants are aware of the collision. Then the packet is retransmitted after $R$ time slots, where $R$ is chosen randomly out of $\{0,1,...,2^j-1\}$ and $j = \min\{i,10\}$. Choosing the retransmission time deterministically would lead to repeated collisions.
  - A similar backoff strategy is used in WLAN.

[Keshav 1997]
Soft State

- State: information that determines future behaviour
- State can be stored in the network (call state in a circuit switched telephone network), it has to be created and removed.
- Incomplete removal leads to problems (e.g. if resources remain reserved). Reacting to all kinds of errors and abnormal terminations can lead to a complicated design.
- **Soft state** can be a solution: State is not persistent, it has to be refreshed (requires bandwidth) and will be removed after timeout; i.e. there is an automatic cleanup after failure.

[Keshav 1997]
Exchanging State Explicitly

- Communicating entities often need to exchange state.
- It is advisable to do this explicitly if possible.
- Example: A file transfer protocol splits packets into segments and transmits it to the receiver. How can the receiver detect packet loss?
  - Implicitly by looking into the payload (requires application layer knowledge)
  - Explicitly by assigning sequence numbers to the packets by the sender.

[Keshav 1997]
Hysteresis

- If a system state depends on a variable value, small fluctuations of this value around the threshold result in frequent state changes. This may lead to undesired behaviour.
- Hysteresis means to apply a state-dependent threshold to prevent oscillations.
- Example: Cellular phones are connected to the base station with the best signal quality. As a handover from one cell to another is an expensive operation, it is only performed if the increase in signal strength is above a certain threshold.

[Keshav 1997]
Separating data and control

- Separating per-path or per-connection actions and per-packet actions.
- Can increase throughput, but requires state information in the network (less robust, cf. ‘distributed state vs. fate sharing’, Chapter 2)
- Example: In *Virtual Circuits* control packets are to set up a connection. Data packets carry only a virtual circuit identifier.

[Keshav 1997]
Many systems obey Pareto’s law or the 80/20 rule:
• Only 20% of the code is used in 80% of the time.

Optimizing these 20% improves the overall performance.

Example: In a certain protocol, most packets to be processed are of the same type. Thus, we should check for this common case first and optimize the code for processing it.

[Keshav 1997]
Extensibility

- Design should allow for future extensions
- Example:
  - The IP packet header contains a version number that indicates the format of the rest of the header.
  - Extension fields in ASN.1 (see Chapter 4.III)

[Keshav 1997]
## Overview

<table>
<thead>
<tr>
<th>What you want</th>
<th>What you can buy</th>
<th>What you have to pay (other than labour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>Parallelism</td>
<td>Computation</td>
</tr>
<tr>
<td></td>
<td>Batching</td>
<td>Larger response time</td>
</tr>
<tr>
<td>Response time</td>
<td>Caching (exploiting locality)</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td>Optimizing the common case</td>
<td></td>
</tr>
<tr>
<td>Reduce bandwidth consumption</td>
<td>Separating data and control</td>
<td>Lack of robustness</td>
</tr>
<tr>
<td>Access to a shared resource</td>
<td>Multiplexing</td>
<td>Shared bandwidth</td>
</tr>
<tr>
<td></td>
<td>Virtualization</td>
<td>Control overhead</td>
</tr>
<tr>
<td>Robustness</td>
<td>Randomization</td>
<td>Control overhead</td>
</tr>
<tr>
<td></td>
<td>Soft state</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hysteresis</td>
<td></td>
</tr>
</tbody>
</table>
Architectural Considerations

▫ Architecture: Decomposition into functional modules.
▫ Common approach: Protocol Layering
  ▪ Each protocol layer defines a level of abstraction
  ▪ Design objective: Integration of related functions with well-defined interfaces
  ▪ cf. Chapter 2, “Modularity”
▫ Alternative Approach: Integrated Layer Processing (ILP)
  ▪ Processing of data in an integrated application layer
  ▪ Goal: Minimizing data access and copy operations (especially in the upper layers)
Integrated Layer Processing (1)

- Motivation: Applications that exchange large amounts of data lose performance when copying data between layers.
- Data is processed sequentially in layered architectures.
- *Data manipulation vs. Transfer control*: Memory access and data manipulation can require more computation than controlling the communication.
- Thus, data manipulation operations should be integrated in the application layer.

## Manipulation vs. Control

<table>
<thead>
<tr>
<th>Layer</th>
<th>Data manipulation</th>
<th>Transfer control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>copying to appl. address space</td>
<td></td>
</tr>
<tr>
<td>Presentation</td>
<td>encryption, formatting</td>
<td></td>
</tr>
<tr>
<td>Session</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transport</td>
<td>buffering for retransmission</td>
<td>congestion control, ACKs detection of transmission problems</td>
</tr>
<tr>
<td>Network</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Link</td>
<td>error detection and correction, buffering</td>
<td>flow control, framing, ACKs</td>
</tr>
<tr>
<td>Physical</td>
<td>network access</td>
<td>Multiplexing</td>
</tr>
</tbody>
</table>

## Integrated Layer Processing (2)

<table>
<thead>
<tr>
<th>ILP</th>
<th>Data manipulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td></td>
</tr>
<tr>
<td>Presentation</td>
<td>Integrated Application Layer</td>
</tr>
<tr>
<td>Session</td>
<td></td>
</tr>
<tr>
<td>Transport</td>
<td>simple datagram protocol, e.g. UDP</td>
</tr>
<tr>
<td>Network</td>
<td></td>
</tr>
<tr>
<td>Data Link</td>
<td></td>
</tr>
<tr>
<td>Physical</td>
<td></td>
</tr>
</tbody>
</table>

**Example:** ILP and Internet Protocols
Integrated Layer Processing (3)

- **ILP Principles:**
  - The application deals with out-of-order transmission or loss of data. It triggers retransmissions.
  - The application splits data into data units (rather than letting the transport layer do the segmentation)
  - Repeated data processing is avoided by using one main processing loop
- ILP targets mainly the avoidance of presentation layer processing

Application Level Framing

- The application splits the data into Application Data Units (ADUs).

- Requirements:
  - The sender can label each ADU such that the receiver can determine its place in the sequence of ADUs.
  - The application must be able to process ADUs out of order. Their size has to be defined accordingly.

Benefits and Limitations of ILP

› **Advantages**
  • No additional presentation conversion
  • Permits efficient implementation of data manipulation

› **Disadvantages**
  • The principles of layering are abandoned
  • ILP implementations can lead to various protocol stack variants (with customized implementations)
    - Losing flexibility and maintainability

Protocol Building Blocks

- Basic methods that most protocols implement or rely on:
  - **Error control**: Detection and correction of transmission errors
  - **Flow Control**: Adaption of the transmission rate to the service rate of the receiver
    (Related: congestion control)
- Both exist as point-to-point or end-to-end mechanisms (in multihop networks)
- They are usually implemented in the link layer and in the transport layer
Error control

- Error control
  - Error detection (e.g. CRC)
  - Error correction
    - Forward Error Correction
    - Backward Error Correction (ARQ schemes)

- see Lecture “Systeme II” for more details.
Types of Errors

- **Bit Errors**
  - Corruption of single bits
  - due to noise, loss of synchronization, etc.
  - Error control typically on the link layer

- **Packet Errors**
  - Loss, duplication and re-ordering
  - Error control typically on the transport layer
Error Control

- **Bit error detection and correction**
  - Basic idea: adding redundant information, e.g. parity codes, hamming codes, CRC

- **Packet error detection and correction**
  - Detection by sequence numbers or handshakes
  - Retransmission
  - Forward Error Correction
Cyclic Redundancy Check (CRC)

- View data bits D as a binary number
- Choose r+1 bit pattern G (generator)
- Idea: choose r CRC bits, R, such that
  - <D,R> exactly divisible by G (modulo 2)
  - receiver knows G, divides <D,R> by G. If non-zero remainder: error detected!

\[
D \cdot 2^r \text{ XOR } R
\]

- widely used in practice (Ethernet, 802.11 WiFi, ATM)

[Kurose, Ross, 2007]
CRC Example

Data: 1101011011
Generator: 10011

Coding:

Decoding:

Decoding with error:
CRC

- **Detects**
  - all single bit errors
  - almost all 2-bit errors
  - any odd number of errors
  - all bursts up to M, where generator length is M
  - longer bursts with probability $2^{-M}$

- **Advantage:**
  - Can be checked on-the-fly with a shift register

[Keshav 1997]
Types of packet errors (1)

- **Loss**
  - due to uncorrectable bit errors
  - due to buffer overflow
    - especially with bursty traffic
    - loss rate depends on burstiness, load, and buffer size
  - fragmented packets can lead to error multiplication
    - the longer the packet, the more the loss

[Keshav 1997]
Types of packet errors (2)

- **Duplication**
  - The same packet is received twice
    (usually due to retransmission)

- **Insertion**
  - Packet from some other conversation received
    - header corruption

- **Reordering**
  - Packets received in wrong order
    - usually due to retransmission
    - some routers also re-order

[Keshav 1997]
Packet error detection and correction

- **Automatic Repeat reQuest (ARQ)**
  - Detection
    - Sequence numbers
    - Timeouts
  - Correction
    - Retransmission

- **Basic ARQ schemes:**
  - Stop-and-Wait, Go-back-N, Selective Repeat

[Keshav 1997]
Sequence numbers

- Sequence numbers are added to each packet header
- Incremented for new (non-retransmitted) packets
- Sequence space
  - set of all possible sequence numbers
  - for a 3-bit seq #, space is \{0,1,2,3,4,5,6,7\}
- Sequence numbers should be long enough so that the sender does not confuse sequence numbers on acks

[Keshav 1997]
Using sequence numbers

- **Loss**
  - Gap in sequence space allows receiver to detect loss
    - e.g. received 0,1,2,5,6,7 => lost 3,4
  - ACKs carry cumulative sequence number
  - Redundant information
  - If no ACK for a while, sender suspects loss

- **Reordering**

- **Duplication**

- **Insertion**
  - If the received seq. number is “very different” from what is expected

[Keshav 1997]
Loss detection

- **By the receiver**, from a gap in sequence space
  - send an NACK to the sender?
  - Disadvantages of NACKs:
    - Extra load in case of loss
    - Moves retransmission problem to the receiver

- **By the sender**, by looking at cumulative ACKs, and after timeout
  - requires to choose timeout interval

[Keshav 1997]
Timeouts

- **Retransmission after timeout:**
  - Set timer on sending a packet
  - If timer goes off, and no ACK received, then retransmit

- **How to choose timeout value?**
  - Intuitively, we expect a reply in about one round trip time (RTT)
  - Static scheme: RTT known a priori
  - Dynamic scheme: RTT measurement

[Keshav 1997]
Retransmissions

- Sender detects loss on timeout
- Which packets to retransmit?
  - Depends on the chosen scheme
  - Typically based on the concept of the error control window

[Keshav 1997]
Stop-and-Wait ARQ

- After each data packet wait for an ACK.
- Retransmit after timeout.
- Simple scheme, easy implementation
- Can be used for flow control as well.

[Keshav 1997]
Error control window

- Set of packets sent, but not acked

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | (original window)
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | (recv ack for 3)
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | (send 8)

- May want to restrict max size = window size
- Sender blocked until ack comes back

[Keshav 1997]
Go-back-N

- On a timeout, retransmit the entire error control window
- Receiver only accepts in-order packets

Advantages:
- simple, no buffer at receiver

Disadvantages:
- can add to congestion, wastes bandwidth

- used in TCP

[Keshav 1997]
Selective Repeat

- Find out which packets were lost, then only retransmit them.

**How to find lost packets?**

- Each ACK has a bitmap of received packets
  - e.g. cum_ack = 5, bitmap = 101
    => received 5 and 7, but not 6
  - wastes header space
- Sender periodically asks receiver for bitmap
- **Fast retransmit**
  - If sender gets the same cumulative ACK repeatedly
  - then retransmit packet with number = cum_ack + 1

[Keshav 1997]
Flow control

- **Open-loop flow control**
  - Description of the traffic by the source upon connection establishment
  - Resource reservation

- **Closed-loop flow control**
  - Dynamic adaption upon feedback
  - There are also combined (hybrid) schemes

- see Lecture “Systeme II” for more details.
Closed-loop Flow Control

› **Example 1:** Stop-and-Wait Protocol

[Keshav 1997]
Closed-loop Flow Control

- **Example 2:** Static-window

[Keshav 1997]
Closed-loop Flow Control

- **Transport layer congestion control** in the Internet:
  - Adaptive window size: In the absence of loss, the window is increased, on loss the size is reduced using the AIMD policy. (TCP-Tahoe and TCP-Reno)

![Diagram of Closed-loop Flow Control](image)

[Keshav 1997]
Implementation Techniques

- Protocol behaviour is modeled by extended finite state machines.
- Standard techniques to transform state machine diagrams or state transition tables into code (cf. Chapter 4.I):
  - Nested switch/case
  - Table-driven
  - State design pattern
- Code generators available for UML/SDL state machines
- Various libraries available that support state transition tables or state machine generation
enum State {q0, q1, q2, ...};
enum Event {e1, e2, ...};

static State s = q0;

void handle(Event e)
{
    switch(s)
    {
    case q0:
        switch(e)
        {
            case e1:
                s = q1;
                break;
            case e2:
                s = q2;
                break;
            [...]
        }
        break;
    case q1:
        switch(e)
        {
            case e1:
                s = q2;
                break;
            case e2:
                s = q0;
                break;
            [...]
        }
        break;
    case q2:
        switch(e)
        {
            case e1:
                s = q1;
                break;
            case e2:
                s = q1;
                break;
            [...]
        }
        break;
    [...]
    }
Table-driven Implementation (1)

- Implementation of a state transition table as a two-dimensional array (mapping states and events to next states).

```c
enum State {s0, s1, s2};
enum Event {e1, e2};

transition[s2+1][e2+1] =
{
  {s1, s2},
  {s2, s0},
  {s0, s1},
};
```

- The next state can be derived as follows:

```c
State changeState(State s, Event e) {
  return transition[s][e];
}
```

Table-driven Implementation (2)

- Switch/case solution for a Mealy machine (transitions have events and actions)

```java
switch (state) {
    case state_0:
        eventHandler_0(event);
        break;
    ...
    case state_N:
        eventHandler_N(event);
        break;
    default:
        error("unexpected state");
}
state = nextState(state, event);
```

```java
void eventHandler_0(Event e) {
    switch(e) {
        case event_0:
            executeAction_0_0;
            break;
        ...
        case event_M:
            executeAction_0_M;
            break;
        default:
            error("unexpected event");
    }
}
```


this should actually be obsolete!
Table-driven Implementation (3)

- Switch/case solution for a Moore machine (transitions have only events, states have entry actions)

```c
next_state = nextState(state, event);
if (state != next_state) {
    state = next_state;
    executeEntryAction(state);
}
```

```c
void executeEntryAction() {
    case state_0:
        executeAction_0;
        break;
    ...
    case state_N:
        executeAction_N;
        break;
    default:
        error("unexpected state");
}
```

Variant with Function Pointers

- Table-driven solution with function pointers:

```c
enum State {s0, s1, s2};
enum Event {e1, e2};
typedef void (*fptr)();
typedef struct StateEntry {
  State nextState;
  fptr action;
}
transition[s2+1][e2+1] = {
  { {s1,a11}, {s2,a22},
    { {s2,a12}, {s0,a20} ,
    { {s0,a10}, {s1,a21} },
  };

global void changeState(State s, Event e) {
    stateEntry se = transition[s][e];
    state = se.nextState;
    (*se.action)();
}
void a10() {
  ...
}
void a11() {
  ...
} ...
```
State Design Pattern

- Object-oriented technique
- The State Design Pattern
  - define an abstract superclass with an event handler and derive a concrete class for each state
  - associate the state with the class holding the context (the state machine)
  - change of behaviour by object change
State pattern

- Each state is represented by a separate class
- State change is performed by instantiating a new object

[Gamma et al., Design Patterns, Addison Wesley, 1994]
State pattern

class Context {
    private State state;
    public void setState(State s) {
        state = s;
    }
    handleEvent(Event e) {
        state.handle(e, this);
    }
}

interface State {
    public void handle(Event e, Context c)
}

class ConcreteState1 implements State {
    public void handle(Event e, Context c) {
        switch (e) {
            case e1: context.setState(new State1);
            break;
            case e2: context.setState(new State2);
            break;
        }
    }
}

class ConcreteState2 implements State {
    [...]
    [...]
}
Table-driven vs. State pattern

- State pattern models state-specific behaviour
  - Transition logic can be implemented in the context class or in the state subclasses
  - Implementing transitions in state subclasses is more flexible, but introduces dependencies!

- Table-driven implementation focuses on state transitions
  - Table lookups are less efficient than function calls

[Gamma et al., Design Patterns, Addison Wesley, 1994]
From FSM to Code

- All states and events should be well-defined

- The state machine or state transition table has to cover all combinations of states and events and define the corresponding action

- This includes also conditions on state transitions

- Don’t forget to optimize the common case!