Presenter: Sandeep K. S. Gupta

Reference:
- Simulation Modeling and Analysis Law and Kelton – Chapter 1 – Basic Simulation Modeling, McGraw Hill.
- Chapter 3, 5 Computer Networking, Kurose and Ross, Addison Wesley
- Chapter 7, An Engineering Approach to Networking, Keshav, Addison Wesley.

Arizona State University
School of computing, informatics, and decision systems engineering
Agenda

- Introduction to Computer Simulation
  - Discrete Event Simulation
  - Example Single Server Queue
  - Advantages, Disadvantages and Pitfalls

- Protocol Specification, Performance Modeling
  - Example: Network Link Layer
    - (Medium) Random access protocol: performance modeling
    - Reliable Data Transfer Protocol: Specification and Performance Modeling
Ways to Study System

- Experiment with the actual system
- Experiment with a model of the system
  - Physical model
  - Mathematical model
    - Analytical solution
    - Simulation
COMPUTER SIMULATION

- **Simulation**: Imitate the operations of a facility or process, usually via computer
  - **System**: what’s being simulated
    - To study system, often make assumptions/approximations, both logical and mathematical, about how it works
  - **Model**: assumptions form a *model* of the system
    - If model structure is simple enough, could use mathematical methods to get exact information on questions of interest — *analytical solution*

- But most complex systems require models that are also complex (to be valid)
  - Must be studied via simulation — evaluate model numerically and collect data to estimate model characteristics

- Example: A company if considering whether to use new job scheduling policy to improve its datacenter’s energy efficiency
  - Implement it and see if the data center becomes more energy efficient?
  - Simulate current and new scheduling technique
Impediments to acceptance, use of simulation

- Models of large systems are usually very complex
  - But now have better modeling software … more general, flexible, but still (relatively) easy to use

- Can consume a lot of computer time
  - But now have faster, bigger, cheaper hardware to allow for much better studies than just a few years ago … this trend will continue
  - However, simulation will also continue to push the envelope on computing power - we ask more and more of our simulation models

- Impression that simulation is “just programming”
  - There’s a lot more to a simulation study than just “coding” a model in some software and running it to get “the answer”
  - Need careful design and analysis of simulation models – simulation methodology
**Types of Systems**

- Types of systems
  - *Discrete*
    - State variables change instantaneously at separated points in time
    - Bank model: State changes occur only when a customer arrives or departs
  - *Continuous*
    - State variables change continuously as a function of time
    - Airplane flight: State variables like position, velocity change continuously

- Many systems are partly discrete, partly continuous
Classification of Simulation Models

- Classification of simulation models
  - Static vs. dynamic
  - Deterministic vs. stochastic
  - Continuous vs. discrete

- *discrete-event simulation models are* dynamic, stochastic, and discrete
Components and Organization of a Discrete-Event Simulation Model

- Each simulation model must be customized to target system
- But there are several common components, general organization
  - *System state* – variables to describe state
  - *Simulation clock* – current value of simulated time
  - *Event list* – times of future events (as needed)
  - *Statistical counters* – to accumulate quantities for output
  - *Initialization routine* – initialize model at time 0
  - *Timing routine* – determine next event time, type; advance clock
  - *Event routines* – carry out logic for each event type
  - *Library routines* – utility routines to generate random variates, etc.
  - *Report generator* – to summarize, report results at end
  - *Main program* – ties routines together, executes them in right order
Components and Organization of a Discrete-Event Simulation Model

1. Initialization routine
   - Set simulation clock = 0
   - Initialize system state and statistical counters
   - Initialize event list

2. Main program
   - Invoke the initialization routine
   - Invoke the timing routine repeatedly

3. Timing routine
   - Determine the next event type, say i
   - Advance the simulation clock

4. Event routine i
   - Update system state
   - Update statistical counters
   - Generate future events and add to event list

5. Is simulation over?
   - No
   - Yes

6. Report generator
   - Compute estimates of interest
   - Write report

7. Stop
STEPS IN A SOUND SIMULATION STUDY

1. Formulate problem and plan the study
2. Collect data and define a model
3. Conceptual model valid?
   Yes
   4. Construct a computer program and verify
5. Make pilot runs
6. Programmed model valid?
   Yes
   7. Design experiments
   8. Make production runs
9. Analyze output data
10. Document, present, and use results
**DISCRETE-EVENT SIMULATION**

- *Discrete-event simulation*: Modeling of a system as it evolves over time by a representation where the state variables change instantaneously at separated points in time
  - state can change at only a *countable* number of points in time
  - These points in time are when *events* occur

- *Event*: Instantaneous occurrence that **may** change the state of the system
  - Sometimes get creative about what an “event” is … e.g., end of simulation, make a decision about a system’s operation

- Can in principle be done by hand, but usually done on computer
Time-Advance Mechanisms

- **Simulation clock**: Variable that keeps the current value of (simulated) time in the model
  - Usually no relation between simulated time and (real) time needed to run a model on a computer
- Two approaches for time advance
  - *Next-event time advance* (usually used)
  - *Fixed-increment time advance* (seldom used)
    - Generally introduces some amount of modeling error in terms of when events *should* occur vs. *do* occur
    - Forces a tradeoff between model accuracy and computational efficiency
Next-event time advance

- Initialize simulation clock to 0
- Determine times of occurrence of future events – *event list*
- Clock advances to next (most imminent) event, which is executed
  - Event execution may involve updating event list
- Continue until stopping rule is satisfied (must be explicitly stated)
- Clock “jumps” from one event time to the next, and doesn’t “exist” for times between successive events … periods of inactivity are ignored
Example: A Simple Processing System

- **Goal:**
  - Estimate expected production
  - Waiting time in queue, queue length, proportion of time machine (server) is busy
Example: Next-Event Advance

Next-event time advance for the single-server queue

\[ t_i = \text{time of arrival of } i\text{th customer (}t_0 = 0) \]

\[ A_i = t_i - t_{i-1} = \text{interarrival time between } (i-1)\text{st and } i\text{th customers (usually assumed to be a random variable from some probability distribution)} \]

\[ S_i = \text{service-time requirement of } i\text{th customer (another random variable)} \]

\[ D_i = \text{delay in queue of } i\text{th customer} \]

\[ c_i = t_i + D_i + S_i = \text{time } i\text{th customer completes service and departs} \]

\[ e_j = \text{time of occurrence of the } j\text{th event (of any type), } j = 1, 2, 3, \ldots \]
Example: Simple Processing System

- Simulation clock variable
- Event calendar: list of event records:
  - [Entity No., Event Time, Event Type]
  - Keep ranked in increasing order on Event Time
  - Next event always in top record
  - Initially, schedule first Arrival, and The End event
- State variables: describe current status
  - Server status $B(t) = 1$ for busy, $0$ for idle
  - Number of customers in queue $Q(t)$
  - Times of arrival of each customer now in queue (a list of random length)
## Simulation by Hand: Setup

<table>
<thead>
<tr>
<th>System</th>
<th>Clock</th>
<th>$B(t)$</th>
<th>$Q(t)$</th>
<th>Arrival times of custs. in queue</th>
<th>Event calendar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of completed waiting times in queue</td>
<td>Total of waiting times in queue</td>
<td>Area under $Q(t)$</td>
<td>Area under $B(t)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$Q(t)$ graph</th>
<th>$B(t)$ graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (Minutes)</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20</td>
</tr>
<tr>
<td>Interarrival times</td>
<td>1.73, 1.35, 0.71, 0.62, 14.28, 0.70, 15.52, 3.15, 1.76, 1.00, ...</td>
</tr>
<tr>
<td>Service times</td>
<td>2.90, 1.76, 3.39, 4.52, 4.46, 4.36, 2.07, 3.36, 2.37, 5.38, ...</td>
</tr>
</tbody>
</table>
Simulation by Hand:  
\( t = 0.00, \) Initialize

<table>
<thead>
<tr>
<th>System</th>
<th>Clock</th>
<th>( B(t) )</th>
<th>( Q(t) )</th>
<th>Arrival times of custs. in queue</th>
<th>Event calendar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>&lt;empty&gt;</td>
<td>[1, 0.00, Arr]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[-, 20.00, End]</td>
</tr>
<tr>
<td>Number of completed waiting times in queue</td>
<td>Total of waiting times in queue</td>
<td>Area under ( Q(t) )</td>
<td>Area under ( B(t) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( Q(t) ) graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Graph of Q(t) with data points]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( B(t) ) graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Graph of B(t) with data points]</td>
</tr>
</tbody>
</table>

Time (Minutes)

Interarrival times 1.73, 1.35, 0.71, 0.62, 14.28, 0.70, 15.52, 3.15, 1.76, 1.00, ...
Service times 2.90, 1.76, 3.39, 4.52, 4.46, 4.36, 2.07, 3.36, 2.37, 5.38, ...
### Simulation by Hand:  
$t = 0.00$, Arrival of Part 1

<table>
<thead>
<tr>
<th>System</th>
<th>Clock</th>
<th>$B(t)$</th>
<th>$Q(t)$</th>
<th>Arrival times of custs. in queue</th>
<th>Event calendar</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>1</td>
<td>0</td>
<td>&lt;empty&gt;</td>
<td>[2, 1.73, Arr]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[1, 2.90, Dep]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[–, 20.00, End]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of completed waiting times in queue</th>
<th>Total of waiting times in queue</th>
<th>Area under $Q(t)$</th>
<th>Area under $B(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**$Q(t)$ graph**

**$B(t)$ graph**

**Interarrival times**: $1.78, 1.35, 0.71, 0.62, 14.28, 0.70, 15.52, 3.15, 1.76, 1.00, ...$

**Service times**: $2.00, 1.76, 3.39, 4.52, 4.46, 4.36, 2.07, 3.36, 2.37, 5.38, ...$
Simulation by Hand:
\( t = 1.73 \), Arrival of Part 2

<table>
<thead>
<tr>
<th>System</th>
<th>Clock</th>
<th>( B(t) )</th>
<th>( Q(t) )</th>
<th>Arrival times of custs. in queue (1.73)</th>
<th>Event calendar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.73</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of completed waiting times in queue</th>
<th>Total of waiting times in queue</th>
<th>Area under ( Q(t) )</th>
<th>Area under ( B(t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>1.73</td>
</tr>
</tbody>
</table>

\( Q(t) \) graph

\( B(t) \) graph

Interarrival times: 1.73, 1.25, 0.71, 0.62, 14.28, 0.70, 15.52, 3.15, 1.76, 1.00, ...

Service times: 2.00, 1.76, 3.39, 4.52, 4.46, 4.36, 2.07, 3.36, 2.37, 5.38, ...
Simulation by Hand:  
\( t = 2.90 \), Departure of Part 1

<table>
<thead>
<tr>
<th>System</th>
<th>Clock</th>
<th>( B(t) )</th>
<th>( Q(t) )</th>
<th>Arrival times of custs. in queue</th>
<th>Event calendar</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.90</td>
<td>1</td>
<td>0</td>
<td>&lt;empty&gt;</td>
<td>[3, 3.08, Arr]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[2, 4.66, Dep]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[−, 20.00, End]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of completed waiting times in queue</th>
<th>Total of waiting times in queue</th>
<th>Area under ( Q(t) )</th>
<th>Area under ( B(t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.17</td>
<td>1.17</td>
<td>2.90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( Q(t) ) graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (Minutes)</td>
</tr>
<tr>
<td>Interarrival times</td>
</tr>
<tr>
<td>Service times</td>
</tr>
</tbody>
</table>
Simulation by Hand: 
\( t = 3.08 \), Arrival of Part 3

<table>
<thead>
<tr>
<th>System</th>
<th>Clock</th>
<th>( B(t) )</th>
<th>( Q(t) )</th>
<th>Arrival times of custs. in queue</th>
<th>Event calendar</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3.08</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of completed waiting times in queue</th>
<th>Total of waiting times in queue</th>
<th>Area under ( Q(t) )</th>
<th>Area under ( B(t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.17</td>
<td>1.17</td>
<td>3.08</td>
</tr>
</tbody>
</table>

\( Q(t) \) graph

\( B(t) \) graph

Interarrival times: 1.73, 1.35, 0.71, 0.62, 14.28, 0.70, 15.52, 3.15, 1.76, 1.00, ...

Service times: 2.90, 1.76, 3.39, 4.52, 4.46, 4.36, 2.07, 3.36, 2.37, 5.38, ...
Simulation by Hand: FAST FORWARD to $t = 20.00$, The End

<table>
<thead>
<tr>
<th>System</th>
<th>Clock</th>
<th>$B(t)$</th>
<th>$Q(t)$</th>
<th>Arrival times of custs. in queue</th>
<th>Event calendar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20.00</td>
<td>1</td>
<td>1</td>
<td>(19.39)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of completed waiting times in queue</th>
<th>Total of waiting times in queue</th>
<th>Area under $Q(t)$</th>
<th>Area under $B(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>15.17</td>
<td>15.78</td>
<td>18.34</td>
</tr>
</tbody>
</table>

**Q(t) graph**

**B(t) graph**

<table>
<thead>
<tr>
<th>Time (Minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 10 15 20</td>
</tr>
</tbody>
</table>

**Interarrival times**

1.73, 1.25, 0.71, 0.62, 14.28, 0.70, 15.52, 3.15, 1.76, 1.00, ...

**Service times**

2.00, 1.76, 3.39, 4.02, 1.16, 4.96, 2.07, 3.36, 2.37, 5.38, ...
Simulation by Hand: Finishing Up

- Average waiting time in queue:
  \[
  \frac{\text{Total of times in queue}}{\text{No. of times in queue}} = \frac{15.17}{6} = 2.53 \text{ minutes per part}
  \]

- Time-average number in queue:
  \[
  \frac{\text{Area under } Q(t) \text{ curve}}{\text{Final clock value}} = \frac{15.78}{20} = 0.79 \text{ part}
  \]

- Utilization of resource:
  \[
  \frac{\text{Area under } B(t) \text{ curve}}{\text{Final clock value}} = \frac{18.34}{20} = 0.92 \text{ (dimension less)}
  \]
OTHER TYPES OF SIMULATION

- **Continuous simulation**
  - Typically, solve sets of differential equations numerically over time
  - May involve stochastic elements
  - Tools such as MATLAB can be used

- **Combined discrete-continuous simulation**
  - Continuous variables described by differential equations
  - Discrete events can occur that affect the continuously-changing variables
  - aka Hybrid models
Other Types of Simulation

- *Monte Carlo simulation*
  - No time element (usually)
  - Wide variety of mathematical problems
  - Example: Evaluate a “difficult” integral $I = \int_a^b g(x) \, dx$
    - Let $X \sim U(a, b)$, and let $Y = (b - a) \, g(X)$
    - Then
      \[
      E(Y) = E[(b - a)g(X)] \\
      = (b - a)E[g(X)] \\
      = (b - a)\int_a^b g(x) \, f_X(x) \, dx \\
      = (b - a)\int_a^b g(x) \, \frac{1}{b - a} \, dx \\
      = \int_a^b g(x) \, dx \\
      = I
      \]
  - Algorithm: Generate $X \sim U(a, b)$, let $Y = (b - a) \, g(X)$; repeat; average the $Y$’s … this average will be an unbiased estimator of $I$
ADVANTAGES, DISADVANTAGES, AND PITFALLS OF SIMULATION

- Advantages
  - allows great flexibility in modeling complex systems
  - easy to compare alternatives
  - control experimental conditions
  - can study system with a very long time frame

- Disadvantages
  - Stochastic simulations produce only estimates – with noise
  - models can be expensive to develop
  - usually produce large volumes of output – need to summarize, statistically analyze appropriately

- Pitfalls
  - Failure to identify objectives clearly up front
  - In appropriate level of detail
  - Inadequate design and analysis of simulation experiments
Agenda

- Introduction to Computer Simulation
  - Discrete Event Simulation
  - Example Single Server Queue
  - Advantages, Disadvantages and Pitfalls

- Protocol Specification, Performance Modeling
  - Example: Network Link Layer
    - (Medium) Random access protocol: performance modeling
    - Reliable Data Transfer Protocol: Specification and Performance Modeling
Note

- Avg. number of retransmissions required to get packet across over a link with packet failure rate $p$ is $1/q$, where $q = 1-p$.
- If bit error probability is $b$ & packet size $n$, $p = 1 - (1-b)^n$.
  - for small $b$, $p$ approx. = $nb$.
  - for large $n$, $p$ is very close to 1 e.g. for $n=1024$ and $b = .01$, $p = .99$.
  - i.e. every packet would require 100 retransmissions on average
  - Error detection and correction techniques are used to bring down this number but still need reliable protocols.
Link Layer Services

- Framing, link access:
  - encapsulate datagram into frame, adding header, trailer
  - channel access if shared medium
  - “MAC” addresses used in frame headers to identify source, dest
    - different from IP address!

- Reliable delivery between adjacent nodes
  - seldom used on low bit error link (fiber, some twisted pair)
  - wireless links: high error rates
    - Q: why both link-level and end-end reliability?
Link Layer Services (more)

- **Flow Control:**
  - pacing between adjacent sending and receiving nodes

- **Error Detection:**
  - errors caused by signal attenuation, noise.
  - receiver detects presence of errors:
    - signals sender for retransmission or drops frame

- **Error Correction:**
  - receiver identifies *and corrects* bit error(s) without resorting to retransmission

- **Half-duplex and full-duplex**
  - with half duplex, nodes at both ends of link can transmit, but not at same time
Where is Link Layer?

- Link layer implemented in "adaptor" (aka NIC)
  - Ethernet card, PCMCIA card, 802.11 card
- Sending side:
  - Encapsulates datagram in a frame
  - Adds error checking bits, rdt, flow control, etc.
- Receiving side
  - Looks for errors, rdt, flow control, etc
  - Extracts datagram, passes to rcving node
  - Adapter is semi-autonomous
  - Link & physical layers
Error Detection

EDC = Error Detection and Correction bits (redundancy)
D = Data protected by error checking, may include header fields

• Error detection not 100% reliable!
  • protocol may miss some errors, but rarely
  • larger EDC field yields better detection and correction
Parity Checking

**Single Bit Parity:**
Detect single bit errors

- d data bits
- parity bit
- 0111000110101011 0

**Two Dimensional Bit Parity:**
Detect and correct single bit errors

\[
\begin{array}{cccccc}
\text{row parity} & \text{d}_{1,1} & \ldots & \text{d}_{1,j} & \text{d}_{1, j+1} \\
\text{d}_{2,1} & \ldots & \text{d}_{2,j} & \text{d}_{2, j+1} \\
\ldots & \ldots & \ldots & \ldots & \ldots \\
\text{d}_{i,1} & \ldots & \text{d}_{i,j} & \text{d}_{i, j+1} \\
\text{d}_{i+1,1} & \ldots & \text{d}_{i+1,j} & \text{d}_{i+1, j+1} \\
\end{array}
\]

- column parity

1 0 1 0 1 1
1 1 1 1 0 0
0 1 1 1 0 1
0 0 1 0 1 0

*no errors*

1 0 1 0 1 1
1 0 1 1 0 0

*parity error*

1 0 1 0 1 1
1 1 1 1 0 1
0 1 1 1 0 1
0 0 1 0 1 0

*correctable single bit error*
Multiple Access Links and Protocols

Two types of “links”:

- **point-to-point**
  - PPP for dial-up access
  - point-to-point link between Ethernet switch and host

- **broadcast** (shared wire or medium)
  - Old-fashioned Ethernet
  - upstream HFC
  - 802.11 wireless LAN
Multiple Access protocols

- single shared broadcast channel
- two or more simultaneous transmissions by nodes: interference
  - collision if node receives two or more signals at the same time

*multiple access protocol*

- distributed algorithm that determines how nodes share channel, i.e.,
  determine when node can transmit
- communication about channel sharing must use channel itself!
  - no out-of-band channel for coordination
Ideal Multiple Access Protocol

Broadcast channel of rate $R$ bps

1. When one node wants to transmit, it can send at rate $R$.
2. When $N$ nodes want to transmit, each can send at average rate $R/N$
3. Fully decentralized:
   - no special node to coordinate transmissions
   - no synchronization of clocks, slots
4. Simple
MAC Protocols: a taxonomy

Three broad classes:

- **Channel Partitioning**
  - divide channel into smaller “pieces” (time slots, frequency, code)
  - allocate piece to node for exclusive use

- **Random Access**
  - channel not divided, allow collisions
  - “recover” from collisions

- **“Taking turns”**
  - Nodes take turns, but nodes with more to send can take longer turns
Random Access Protocols

- When node has packet to send
  - transmit at full channel data rate R.
  - no a priori coordination among nodes

- two or more transmitting nodes \( \rightarrow \) “collision”,

- random access MAC protocol specifies:
  - how to detect collisions
  - how to recover from collisions (e.g., via delayed retransmissions)

- Examples of random access MAC protocols:
  - ALOHA
  - slotted ALOHA
  - CSMA, CSMA/CD, CSMA/CA
Slotted ALOHA

**Assumptions**
- all frames same size
- time is divided into equal size slots, time to transmit 1 frame
- nodes start to transmit frames only at beginning of slots
- nodes are synchronized
- if 2 or more nodes transmit in slot, all nodes detect collision

**Operation**
- when node obtains fresh frame, it transmits in next slot
- no collision, node can send new frame in next slot
- if collision, node retransmits frame in each subsequent slot with prob. $p$ until success
Slotted ALOHA

Pros
- single active node can continuously transmit at full rate of channel
- highly decentralized: only slots in nodes need to be in sync
- simple

Cons
- collisions, wasting slots
- idle slots
- nodes may be able to detect collision in less than time to transmit packet
- clock synchronization
Slotted Aloha efficiency

**Efficiency** is the long-run fraction of successful slots when there are many nodes, each with many frames to send.

- Suppose N nodes with many frames to send, each transmits in slot with probability \( p \).
- prob that node 1 has success in a slot = \( p(1-p)^{N-1} \).
- prob that any node has a success = \( Np(1-p)^{N-1} \).

For max efficiency with N nodes, find \( p^* \) that maximizes \( Np(1-p)^{N-1} \).
For many nodes, take limit of \( Np^*(1-p^*)^{N-1} \) as N goes to infinity, gives \( 1/e = 0.37 \).

At best: channel used for useful transmissions 37% of time!
Pure (unslotted) ALOHA

- unslotted Aloha: simpler, no synchronization
- when frame first arrives
  - transmit immediately
- collision probability increases:
  - frame sent at $t_0$ collides with other frames sent in $[t_0-1, t_0+1]$
Pure Aloha efficiency

\[ P(\text{success by given node}) = P(\text{node transmits}) \cdot \]

\[ P(\text{no other node transmits in } [p_{0-1},p_0]) \cdot \]

\[ P(\text{no other node transmits in } [p_{0-1},p_0]) = p \cdot (1-p)^{N-1} \cdot (1-p)^{N-1} \]

\[ = p \cdot (1-p)^{2(N-1)} \]

… choosing optimum p and then letting n -> infty …

\[ = \frac{1}{2e} = .18 \]

Even worse!
ALOHA - summary

- Slotted ALOHA simple way to double ALOHA’s capacity
- Make sure transmissions start on a slot boundary
- Halves *window of vulnerability*
- Used in cellular phone uplink
ALOHA schemes summarized
Reliable Data Transfer

- Layering – Building reliable data transfer service over unreliable link
- Specification – using Finite State Machine
    - Equivalent to event-based code
- Correctness
- Performance Modeling
Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Reliable data transfer: service interface

**rdt_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer.

**ndt_send()**: called by rdt, to transfer packet over unreliable channel to receiver.

**deliver_data()**: called by rdt to deliver data to upper.

**rdt_rcv()**: called when packet arrives on rcv-side of channel.
Reliable data transfer:

- Protocol requirement:
  - In-order delivery of packet sent from sender to receiver without duplication

- Consider only unidirectional data transfer
  - But control info will flow on both directions!

- We’ll incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
  - Rdt1.0: reliable data transfer on reliable link
  - Rdt2.0: rdt over link with only bit errors
  - Rdt3.0: rdt with both bit errors and packet loss
Protocol Specification Using FSM

- use finite state machines (FSM) to specify sender, receiver functionality
- FSM specification can be used for
  - establishing protocol correctness
  - Automatic synthesis of (event-driven) code
    - While (true) { event-1: event-handling code for event 1; ...; event-n: event-handling-code for event n; }

Arrow denotes start state

state: when in this “state” next state uniquely determined by next event

Denote a self-loop
Rdt1.0: **reliable transfer over a reliable channel**

- **underlying channel perfectly reliable**
  - no bit errors
  - no loss of packets
- **separate FSMs for sender, receiver:**
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

```
# sender
Wait for call from above

dataset = make_pkt(data)
udt_send(packet)

# receiver
Wait for call from below

dataset = rdt_rcv(packet)
extract (packet, data)
deliver_data(data)
```

sender

receiver
Rdt2.0: **channel with bit errors**

- underlying channel may flip bits in packet
- how to recover from errors?
  - **acknowledgements (ACKs):** receiver explicitly tells sender that pkt received OK
  - **negative acknowledgements (NAKs):** receiver explicitly tells sender that pkt had errors
    - sender retransmits pkt on receipt of NAK
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection: checksum to detect bit errors
  - receiver feedback: control msgs (ACK,NAK) rcvr-\textgreater{}sender
**rdt2.0: FSM specification**

**sender**

- `rdt_send(data)`
- `snkpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`

**Wait for call from above**

**Wait for ACK or NAK**

- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `∧`

**receiver**

- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`

- `udt_send(ACK)`

- `udt_send(NAK)`

- `extract(rcvpkt, data)`
- `deliver_data(data)`

- `udt_send(ACK)`

- `corrupt(rcvpkt)`

- `notcorrupt(rcvpkt)`

- `Wait for call from below`
Protocol testing

- Exercise the protocol FSM for various scenarios
  - Error-free
  - Error scenario
    - Data packet error
    - ....
**rdt2.0: operation with no errors**

- `rdt_send(data)`
- `sndpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`

Wait for call from above

Wait for ACK or NAK

- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`

Wait for ACK or NAK

- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `udt_send(sndpkt)`

Wait for call from above

- `Lambda`

Wait for call from below

- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`
- `udt_send(NAK)`

Wait for call from below

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`
rdt2.0: error scenario

```
rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
```

Wait for call from above

```
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
```

Wait for ACK or NAK

```
rdt_send(sndpkt)
```

```
rdt_send(NAK)
```

```
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
```

Wait for call from below

```
extract(rcvpkt, data)
deliver_data(data)
```

```
udt_send(ACK)
```

```
Lambda
```

```
rdt_rcv(rcvpkt) && isACK(rcvpkt)
```

```
Lambda
```

```
rdt_send(data)
```

```
Lambda
```
rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?
- sender doesn’t know what happened at receiver!
- can’t just retransmit: possible duplicate

Handling duplicates:
- sender retransmits current pkt if ACK/NAK garbled
- sender adds sequence number to each pkt
  - Only two are used 0 and 1
- receiver discards (doesn’t deliver up) duplicate pkt
  - Sequence numbers of delivered pkts: 0,1,0,1,0…….

stop and wait
Sender sends one packet, then waits for receiver response
rdt2.1: sender, handles garbled ACK/NAKs

\[
\begin{align*}
\text{rdt}_\text{send}(\text{data}) \\
\text{sndpkt} &= \text{make}_\text{pkt}(0, \text{data}, \text{checksum}) \\
\text{udt}_\text{send}(\text{sndpkt})
\end{align*}
\]

Wait for call 0 from above

\[
\begin{align*}
\text{rdt}_\text{rcv}(\text{rcvpkt}) &\land \\
&\text{notcorrupt(}\text{rcvpkt} ) \land \\
&\text{isACK(}\text{rcvpkt} )
\end{align*}
\]

Wait for ACK or NAK 0

\[
\begin{align*}
\text{udt}_\text{send}(\text{sndpkt})
\end{align*}
\]

\[
\begin{align*}
\text{rdt}_\text{rcv}(\text{rcvpkt}) &\land \\
&\text{notcorrupt(}\text{rcvpkt} ) \land \\
&\text{isACK(}\text{rcvpkt} )
\end{align*}
\]

Wait for call 1 from above

\[
\begin{align*}
\text{udt}_\text{send}(\text{sndpkt})
\end{align*}
\]

\[
\begin{align*}
\text{rdt}_\text{send}(\text{data}) \\
\text{sndpkt} &= \text{make}_\text{pkt}(1, \text{data}, \text{checksum}) \\
\text{udt}_\text{send}(\text{sndpkt})
\end{align*}
\]

Wait for ACK or NAK 1
rdt2.1: receiver, handles garbled ACK/NAKs

- \( rdt_{rcv}(rcvpkt) \) && (corrupt(rcvpkt))
  - \( sndpkt = make_pkt(NAK, \text{checksum}) \)
  - \( udt_{send}(sndpkt) \)
- \( rdt_{rcv}(rcvpkt) \) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
  - \( sndpkt = make_pkt(ACK, \text{checksum}) \)
  - \( udt_{send}(sndpkt) \)

- Wait for 0 from below
- Wait for 1 from below

- \( extract(rcvpkt, data) \)
- \( deliver_{data}(data) \)
- \( sndpkt = make_pkt(ACK, \text{checksum}) \)
- \( udt_{send}(sndpkt) \)
rdt2.1: discussion

Sender:
- seq # added to pkt
- two seq. #'s (0,1) suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “current” pkt has 0 or 1 seq. #

Receiver:
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can not know if its last ACK/NAK received OK at sender
rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must explicitly include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*
**rdt2.2: sender, receiver fragments**

Sender FSM fragment:
- `rdt_send(data)`
- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`
- `rdt_send(data)`

Receiver FSM fragment:
- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))`
- `udt_send(sndpkt)`
- `extract(rcvpkt,data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK1, checksum)`
- `udt_send(sndpkt)`
New assumption:
underlying channel can also lose packets (data or ACKs)
  - checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach: sender waits "reasonable" amount of time for ACK
  - retransmits if no ACK received in this time
  - if pkt (or ACK) just delayed (not lost):
    - retransmission will be duplicate, but use of seq. #s already handles this
    - receiver must specify seq # of pkt being ACKed
  - requires countdown timer
**rdt3.0 sender**

- **Wait for call 0 from above**
  - rdt_rcv(rcvpkt)
  - Δ
  - udt_send(sndpkt)
  - start_timer

- **Wait for ACK0**
  - rdt_rcv(rcvpkt)
  - & & notcorrupt(rcvpkt) & & isACK(rcvpkt,1)
  - stop_timer

- **Wait for call 1 from above**
  - rdt_rcv(rcvpkt)
  - & & notcorrupt(rcvpkt) & & isACK(rcvpkt,0)
  - stop_timer

- **Wait for ACK1**
  - rdt_rcv(rcvpkt)
  - Δ
  - udt_send(sndpkt)
  - start_timer

- **Rdt_send(data)**
  - sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)
  - start_timer

- **Rdt_rcv(rcvpkt)**
  - & & (corrupt(rcvpkt) || isACK(rcvpkt,1))

- **Timeout**
  - udt_send(sndpkt)
  - start_timer

- **Rdt_send(data)**
  - sndpkt = make_pkt(1, data, checksum)
  - udt_send(sndpkt)
  - start_timer

- **Stop timer**
rdt3.0 in action

(a) operation with no loss

(b) lost packet
rdt3.0 in action

(c) lost ACK

(d) premature timeout
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

\[ T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**9} \text{ b/sec}} = 8 \text{ microsec} \]

- \( U_{\text{sender}} \): utilization - fraction of time sender busy sending

\[ U_{\text{sender}} = \frac{L / R}{\text{RTT} + L / R} = \frac{.008}{30.008} = 0.00027 \]

- 1KB pkt every 30 msec \( \rightarrow \) 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!
rdt3.0: stop-and-wait operation

\[ U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30.008} = 0.00027 \]
Pipelined protocols

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts
- range of sequence numbers must be increased
- buffering at sender and/or receiver

(a) a stop-and-wait protocol in operation
(b) a pipelined protocol in operation
Pipelining: increased utilization

- First packet bit transmitted, $t = 0$
- Last bit transmitted, $t = L / R$
- First packet bit arrives
- Last packet bit arrives, send ACK
- Last bit of 2nd packet arrives, send ACK
- Last bit of 3rd packet arrives, send ACK
- ACK arrives, send next packet, $t = RTT + L / R$

$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.0008$

Increase utilization by a factor of 3!