ELEC 372
Engineering Software Structures

Week 8: Software Architecture

Lecture 1

Software Architecture

- Decomposing system into set of components, connectors
  - Typically coarse grain
- Design tool
  - Helps communicate proposed structure before implementation
  - Serves as basis for dividing work packages
  - Supports analysis of implementation properties

Software Architecture

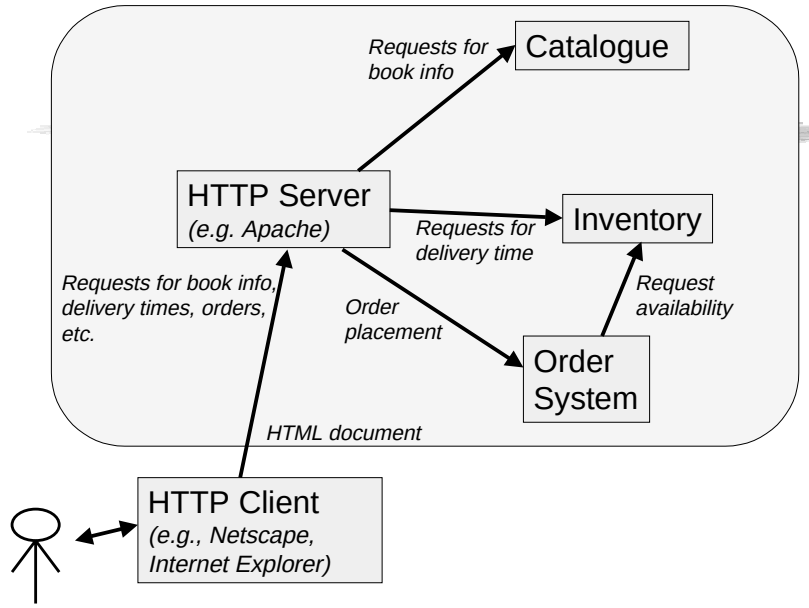
- Architecture Styles, Reference Architectures
  - Encode design knowledge for reuse in other contexts
- Case studies
  - Microsoft COM, OLE
  - Interactive applications with the PAC architecture

Micro-Architectures

- Software architectures within a single component of system architecture
- Object-oriented design
  - Structuring technique for *programming in the large*
- Design patterns
  - Encode OO design solutions with known properties
  - Equivalent of architecture style for micro-architectures

Very Simple Software Architecture Example

- An e-commerce system supporting ordering of books on the WWW
- Customers use web browser to find books, read about books, make orders
- Inventory subsystem can estimate time before book is sent
  - Some books in inventory, some need to be ordered from publisher
- After order is placed, sent to shipping subsystem to be filled
- *For an example of such a system, see http://www.amazon.com*



Components and Connectors

Component

A software or hardware entity implementing some part of the system's functionality

Connector: a mechanism allowing components to communicate. May be a protocol, an operating system mechanism or a programming language feature. Connectors may themselves involve complex code.

Example Connectors

HTTP Connector: source component issues HTTP requests, target component responds with HTML document

CGI Connector: source component calls target component; target component responds with HTML document

Java Call Connector: source component calls method of target component using standard Java method calling conventions; target component returns some Java type

Analyzing Architectures

- Some interesting questions to ask of this architecture
  - *Availability:* What are risks of system not being available (e.g., going down?)
    - Redundant servers?
  - *Security:* What are risks of (e.g.) credit card information being intercepted
    - Perhaps use HTTPS connector instead of HTTP connector?
  - *Performance:* What are risks in case of many users?
    - Perhaps use array of servers, with server redirection?
  - *Modifiability:* What if we want to add a WAP telephone ordering service?

Attribute-Driven Design

- Use quality attributes to motivate design decisions
- Analyze architectural choices based on software architecture chosen

Review of Quality Attributes

- Recall
  - Requirements analysis determines desired quality attributes of system
  - When designing software architecture, attempt to
    - Identify areas of risk where quality attributes may not be met
    - Reduce risk through architectural choices
  - Some attributes amenable to mathematical expression (performance, availability), others less so (modifiability)

Attributes

- Runtime
  - Performance, Security, Availability
- Development Time
  - Modifiability, Portability, Reusability, Testability

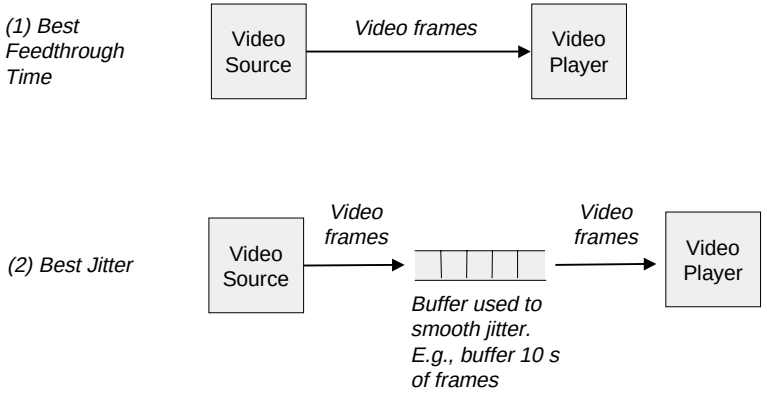
Performance

- Can be measured in terms of
  - Feedback time
    - Time it takes system to respond to user's actions
    - For highly interactive tasks (e.g., drawing a box in PowerPoint), typical requirement = 100 ms
    - For less interactive tasks (e.g., clicking a button), typical requirement = 1 s
    - For computational tasks may be longer (e.g., time to load a web page might be acceptable at 10 s)
  - Feedthrough time
    - Time until other users can see result of one user's actions
    - For highly interactive tasks (e.g., collaborative drawing, Internet telephone), typical requirement = 300 ms
    - Important to synchronize modalities (e.g., sound + gesturing)

Performance

- Can be measured in terms of
  - Jitter
    - Variance in feedback time
      - E.g. in command-line systems such as Unix, low variance in time to complete commands may be more important than feedback time itself
    - Variance in delivery of continuous media such as video
      - Broadcast frame rate = 30 frames per second
      - Therefore frames delivered at 33 ms intervals
      - For video played over Internet (e.g., video on CBC web site), delivery of frames from remote source may not be at such an even rate

Buffering: Trading off Feedthrough Time and Jitter



Question: Which is better for an IP telephone? Which is better for viewing the CBC news?

Performance

- Can be measured in terms of
  - Turnaround time
    - Average time to complete a batch job (e.g., a database query)
  - Throughput
    - Average number of jobs / requests processed per time unit; e.g., number of database queries per second; number of polygons rendered per second)

Availability

- Intuitively represents how much of the time the system is available for use
- System may cease to be available through
  - Network failure
  - Computer hardware failure (e.g., bad memory, disk crash)
  - Software failure
    - Failure of application or operating system
    - Failure of application components to correctly interoperate

Availability

- May be measured as

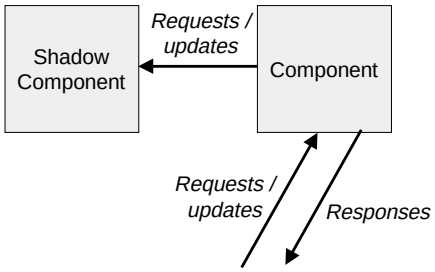
$$\frac{\textit{Mean time to failure}}{\textit{Mean time to failure} + \textit{Mean time to repair}}$$

Mechanisms for Improving Availability

- May address mean time to failure, mean time to repair or both

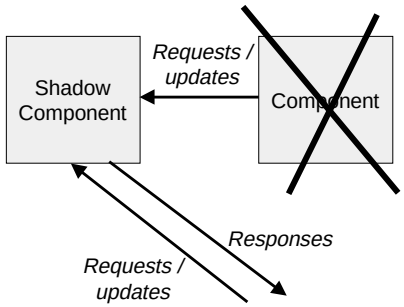
Component Redundancy

- A component responds to requests/updates from other sources
- Forwards all requests/updates to shadow component (e.g., on different machine)
- If component fails, clients reconnect to shadow component and keep going



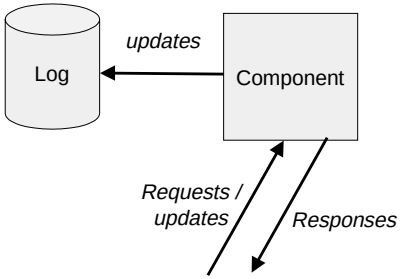
Component Redundancy

- A component responds to requests/updates from other sources
- Forwards all requests/updates to shadow component (e.g., on different machine)
- If component fails, clients reconnect to shadow component and keep going



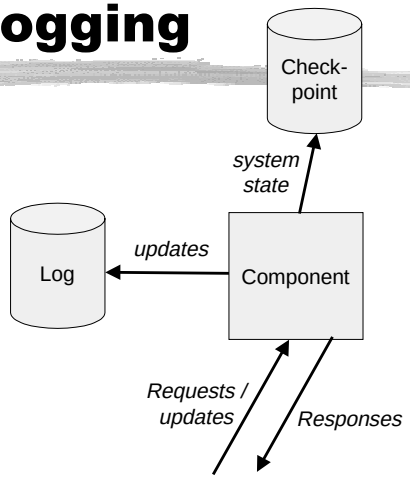
Checkpointing/Logging

- If component fails, need to retrieve its state when it is restarted
- *Checkpointing:* periodically write component state to disk
- *Logging:* write updates to file on disk; can replay updates when component restarted
- Can combine checkpointing and logging - e.g., log goes back to last checkpoint



Checkpointing/Logging

- If component fails, need to retrieve its state when it is restarted
- *Checkpointing:* periodically write component state to disk
- *Logging:* write updates to file on disk; can replay updates when component restarted
- Can combine checkpointing and logging - e.g., log goes back to last checkpoint



ELEC 372
Engineering Software Structures

Week 8: Software Architecture

Lecture 2

Architecture Styles

- Architectural design is a creative process
- Balance tradeoffs between quality attributes of solution
  - Availability, Modifiability, Security, Performance, Cost to implement, …
- When designers create a solution to an architectural problem, it is interesting to record this solution
  - Others can reuse architecture without having to experiment
  - Properties of solution well-understood
  - May help in code reuse

Architecture Styles

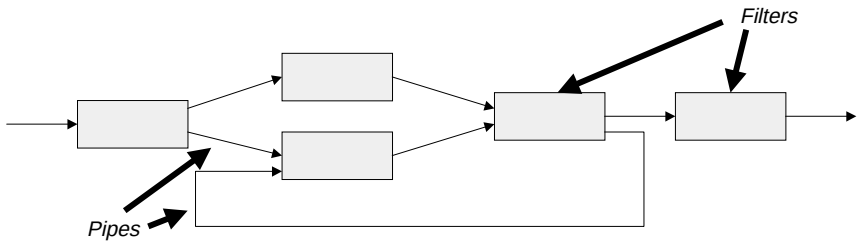
- An architecture style is a set of rules of how to organize a system into a set of components and connectors
- Should be accompanied by
  - *Syntax:* explanation of how to combine components and connectors
  - *Domain description:* explanation of where this style should be applied
  - *Reasoning framework:* explanation of properties of this architecture
- In practice, the reasoning frameworks for architecture styles are often poorly developed

Examples [Shaw&Garlan]

- Pipes and Filters
- Layers
- Event-Based
- Data-Centered
- Interpreter
- PAC

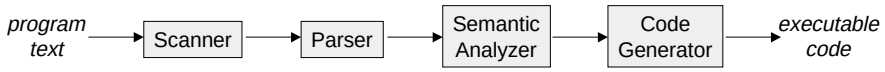
Pipes and Filters Architecture Style

- Synopsis
  - Divides system into set of data *filters*
  - Filters are applied in sequence
  - Data is passed from one filter to the next via *pipes*



Example

- Compiler architecture
  - Compilers map program text (e.g., Java text files) into executable code (e.g., Java byte code, or *".exe"* file containing 80*86 code)
  - This is too complex to do all at once. To simplify structure, pipe and filter architecture often used



Steps in compiling

- Scanning
  - Turn program text into a sequence of *tokens*
  - Tokens represent keywords of language
    - Typically represented as integer constants
  - E.g., **`if (x==3) { … }`** might become
    - tIf tIdentifer "x" tEqualsEquals tIntLiteral 3 …
  - Tokens are actually symbolic constants
    - E.g., tIf=1, tIdentifer=2, tEqualsEquals=3, tIntLiteral=4, giving:
    - 1 2 "x" 3 4 3 …

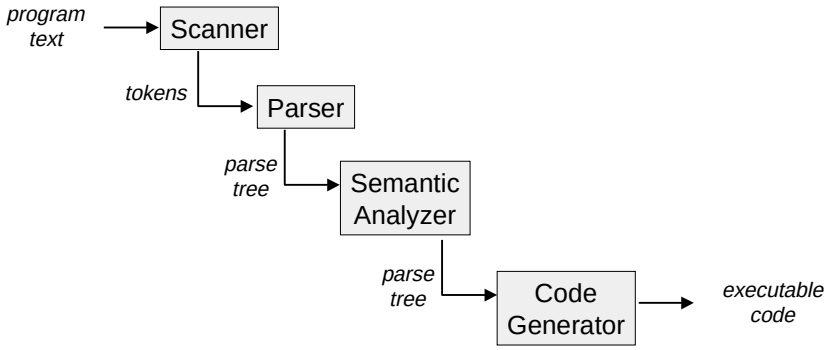
Steps in compiling

- Parsing
  - Check context-free syntax of program
    - E.g., **`if {x==10}`** … generates syntax error
      - I.e., used {…} instead of (…)
  - Generate tree form for expressions
    - E.g.,
      - tIf tIdentifer "x" tEqualsEquals tIntLiteral 3
    - becomes
      - tIf tEqualsEquals tIdentifer "x" tIntLiteral 3

Steps in compiling

- Semantic analysis
  - Check context-sensitive syntax
    - E.g., variables have been declared, types are ok
- Code generation
  - Create native code from parse tree

Compiling as pipe and filter architecture



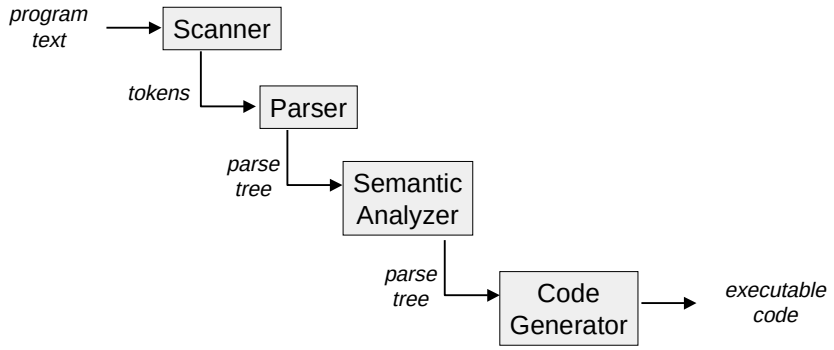
Domain

- Pipe and filter is most appropriate when
  - The problem can be naturally divided into a sequence of processing steps over some data stream
  - The format of the data being passed can be described in a simple and precise manner
  - The complexity of the problem is in the processing steps

Properties of Pipe and Filter

- Modifiability
  - Modifications to system may impact three areas
    - *Change filter:* e.g., implement faster scanning algorithm
      - Change localized to filter itself, which is good
    - *Change format of data passed between filters:* e.g., add a new token, change parse tree format
      - Expensive, since (at least) two filters impacted by change
    - *Add a new filter:* inexpensive if data format not impacted
    - Therefore modifiability properties good if data format stable

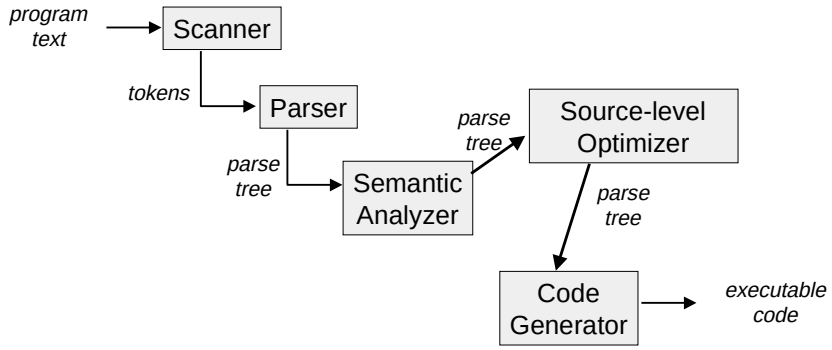
E.g., adding an optimizer to the compiler



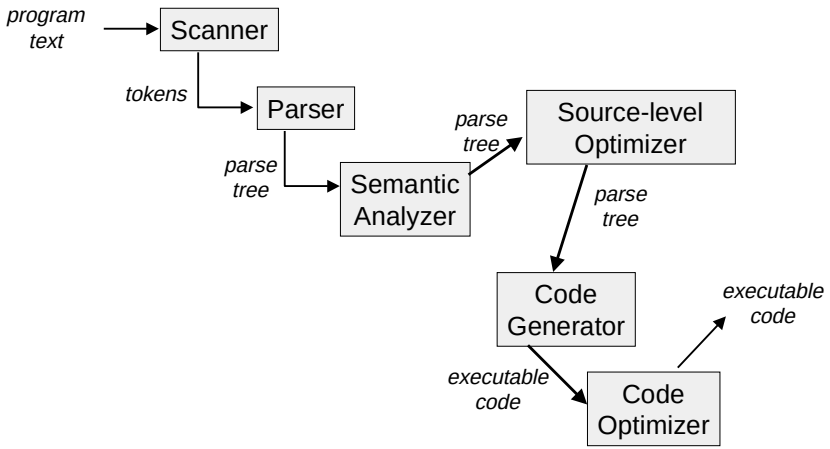
Adding an optimizer to the compiler

- Source-level optimizer improves source code
  - `while (x>3) { x=x-1; if (x>5) {s}; }`
  - improved to
    - `while (x>6) {x=x-1; s} if (x>3) {x=3;}`
- Code-level optimizer improves executable code

E.g., adding an optimizer to the compiler



E.g., adding an optimizer to the compiler

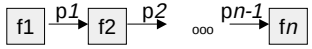


Properties of Pipe and Filter

- Performance
  - Some opportunity for parallel processing
    - Run each filter on a different processor
    - Only works if filters work incrementally -- I.e., filters are not obliged to wait for preceding filter to complete its work before starting themselves
    - Performance gains will only be significant if parallel filters each perform significant work

Properties of Pipe and Filter

- Simple model to analyze parallel performance:



- Let t1,…t*n* be the time to compute each filter f1,…fn
- Let s1,…s*n-1* be the time to move data over the pipes p1,…p*n-1* in the sequential case
- Let d1,…d*n-1* be the time to move data over the pipes p1,…,d*n-1* in the parallel case
- Then execution time in the sequential case is
  - $\sum_{i=1}^{n} t_i + \sum_{i=1}^{n-1} s_i$
- and execution time in the parallel case is
  - $\max_{i=1}^{n} f_i + \sum_{i=1}^{n-1} d_i$

Note simplifying assumption: all filters are able to start processing data at the same time. Can you think of cases where this might not be true?

Properties of Pipe and Filter

- Performance (2)
  - Some optimizations may be lost by strict decomposition into filters
  - There may be some overhead of creating similar data structures from streams provided by pipes
  - E.g.,
    - parser creates parse tree
    - turns parse tree into token stream to send over pipe
    - semantic analyzer recreates parse tree from token stream

ELEC 372
Engineering Software Structures

Week 8: Software Architecture

Lecture 3

Architecture Styles

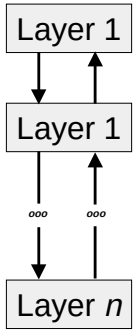
- An architecture style is a set of rules of how to organize a system into a set of components and connectors
- Should be accompanied by
  - *Syntax:* explanation of how to combine components and connectors
  - *Domain description:* explanation of where this style should be applied
  - *Reasoning framework:* explanation of properties of this architecture
- In practice, the reasoning frameworks for architecture styles are often poorly developed

Examples [Shaw&Garlan]

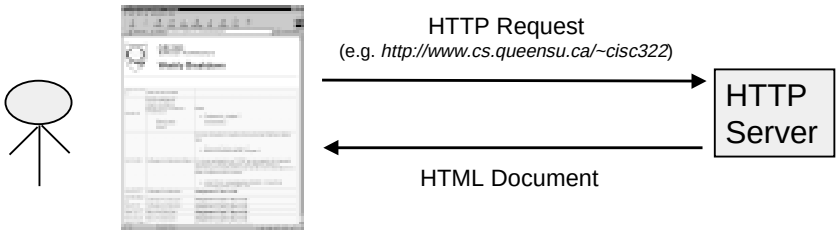
- Pipes and Filters
- Layers
- Event-Based
- Data-Centered
- Interpreter
- PAC

Layers Architecture Style

- System divided into a set of layers
- Each layer is a component
- Each component allowed to use services of layer *below* it
- Provides interface to layer *above* it
  - Set of methods



Example: Web Browsing



-- *Web Browser* issues *HTTP Request* (a string)

-- *HTTP Server* responds with *HTML Document* (also a string)

-- *Web Browser* renders *HTML Document* as a web page

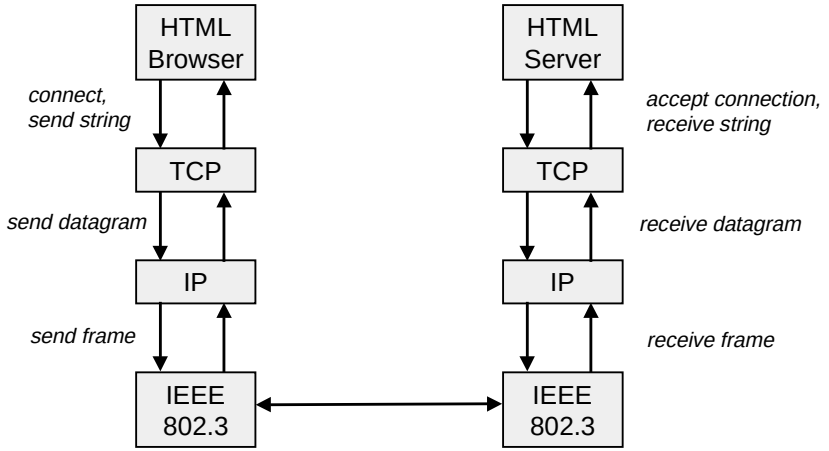
Example HTTP Document

```
<h2><font
face="Arial,Helvetica">Text</font></h2>
The course text is:
<ul>Bernd Oestereich, <i>Developing Software
with UML, </i>Addison-Wesley,
1999</ul>
For those interested in further reading, I
recommend:
<blockquote>Len Bass, Paul Clements and Rick
Kazman, <i>Software Architecture in
Practice,</i> Addison-Wesley, 1998
<p>Mary Shaw and David Garlan, <i>Software
Architecture: Perspectives on
an Emerging Discipline,</i> Prentice Hall, 1996
<p>Erich Gamma, Richard Helm, Ralph Johnson
and John Vlissides, <i>Design
Patterns: Elements of Reusable Object-Oriented
Software,</i> Addison-Wesley,
1995</blockquote>
```



A fragment of an HTML document and how it looks in a browser

Web Browsing as a Layered Architecture

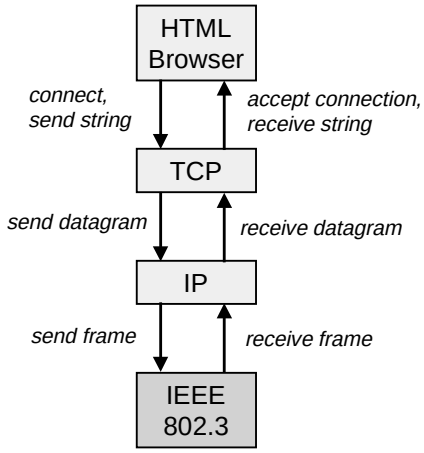


Web Browsing as a Layered Architecture

- Layers in protocol stack communicate by
  - providing information to layer above
  - calling layer below
- Layers provide increasingly abstract view of the network
  - the higher the layer, the more abstract the view
- Layers encapsulate functionality that can be plug-replaced

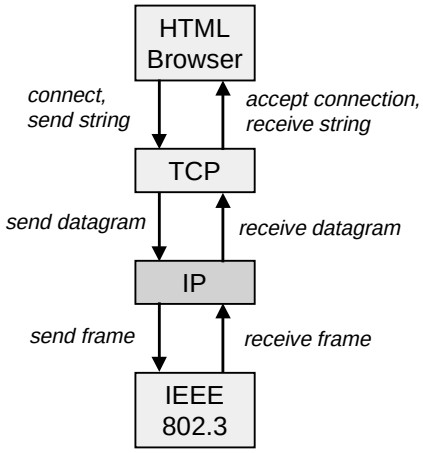
Link Layer

- Level that interfaces directly with the hardware
  - *IEEE 802.3* protocol for Ethernet
  - *PPP* for modems
  - …etc.
- Delivery not guaranteed
  - Sent frames/packets may or may not arrive
- Routing
  - Need machine address of recipient
- Over complete connection, multiple protocols usually required
  - E.g., Modem to backbone to local area Ethernet

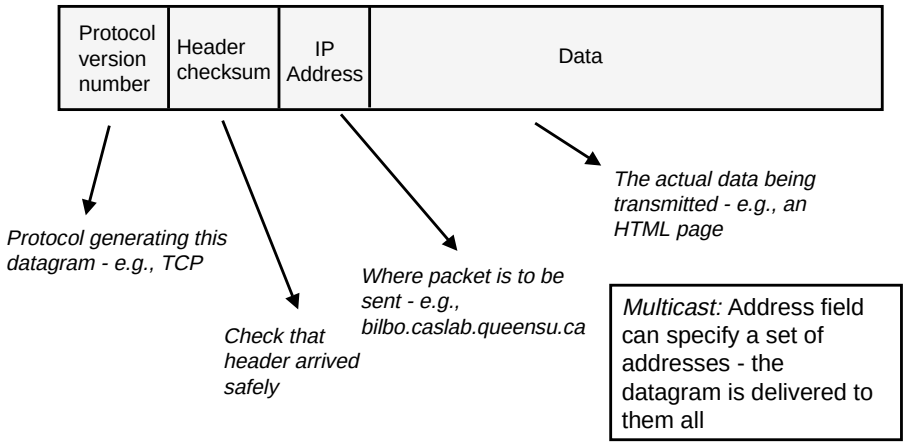


Internet Protocol Layer

- Internet protocol (IP) provides:
  - *datagram abstraction:* message plus address
  - *send/receive:* operations to send/receive a datagram
- Delivery not guaranteed:
  - If datagram contains message "hello world", receiver may receive exactly this string, the string may get lost, or may even be received twice!
- Routing:
  - like a letter, datagram includes address - no connection

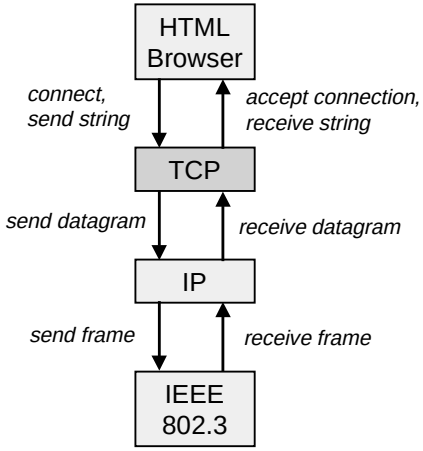


IP Datagram (simplified)



Transport Layer

- Transmission control protocol (TCP) provides:
  - *socket connections:* like notion of dialing a telephone, establishes a connection between two parties over which data can be sent
  - *send/receive:* operations to send/receive a string value
- Delivery is guaranteed:
  - If browser sends string "www.cs.queensu.ca", receiver receives exactly this string

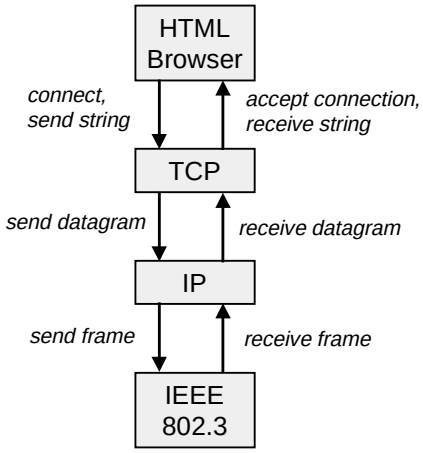


TCP

- TCP breaks messages into set of IP datagrams
  - Adds address to each datagram
- Verifies that all datagrams arrive
  - Discards any duplicates
- When message completely reassembled, makes available to application layer

Summary

- Each layer presents a different abstraction of the network
- Higher-level layers make the network easier to use
  - Routing, lossless delivery
- Each layer
  - provides services to the layer above it
  - uses services in the layer below it



Properties of the Layers Architecture

- Performance
- Modifiability

Performance

- If we are developing layer k, we are limited to the services of layer k+1
- Possible that lower layers have services that might be important to performance
- Therefore trade off advantages in separation of concerns, modifiability versus possible performance issues

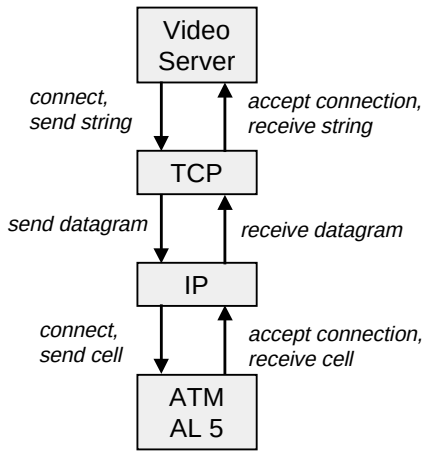
Performance Example

- Imagine we wish to transmit audio/video data
- Video frames encoded as messages sent via TCP
  - E.g., MPEG, Motion JPEG, AVI, QuickTime,…
- High variance in reception of packets leads to "jitter" in video
  - Some retries may be required if video lost
- Better to have "lossy" protocol where some frames may be lost, but frames can be played as soon as they arrive



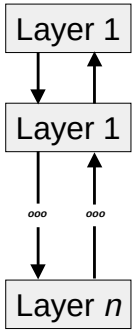
Video over ATM

- ATM (Asynchronous Transfer Mode) is mainly used to implement backbone networks
- ATM is connection-oriented
- Supports multimedia (video and sound) through *quality of service* (QOS) guarantees
  - Reserve part of the network for this connection
- Architectural mismatch with IP layer
  - Ability to create connections with guaranteed QOS would be useful to Video Server



Modifiability

- If layer *k* changed internally
  - No impact on other layers
- If services provided by layer *k* change
  - Impact limited to layer *k-1*
  - I.e., only layer *using* layer *k* may have to change



Modifiability Questions

- If you are considering implementing a system based on layers, need to ask
  - What are likely modifications?
  - Can these modifications be confined to the layers?
  - Do they require changes to multiple layers?

Modifying Layers: Better Support for Video

- Standard solution is to replace TCP component with component providing lossy protocol
- UDP *(user datagram protocol)* is a standard for sending video data over the Internet
- UDP Services are lower level than TCP services
  - Therefore requires modification at application layer

