Software Systems Architecture

FEUP-M.EIC-ASSO-2023-2024

Ademar Aguiar, Neil Harrison

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Quality Attributes (Non-functional attributes)

“A non-functional property of a software system is a constraint on the manner in which the system implements and delivers its functionality”

Functional attributes (features): the WHAT

Non-functional attributes (quality attributes): the HOW (how the system behaves)

Typical Quality Attributes

Security

Reliability

Availability

Efficiency

Performance

Scalability

Fault-tolerance

Usability

Portability

Internal:

- Complexity
- Implementability
- Maintainability

External or Internal?

Some Quality Attributes have direct impact on the user of the system (External)

Some are invisible to the user; but important to the developers (internal)

Sometimes called "Run time" and "Design time" (the Bass book)

The <u>external</u> ones are generally easier to characterize and quantify.

What are Quality Attributes

Group interactive activity:

You are wanting to buy a new car.

Name as many features of the car as you can

- Let's group them by:
  - Functional attributes
  - Non-functional attributes

How do you characterize functional attributes versus non-functional (quality) attributes?

Measuring Quality Attributes

Functional Attributes are often binary measures:

- Does the car have door handles?
- Does the car have 17 cup holders?
- How many seats does it have? (binary: does it have four seats?)

Quality Attributes are usually a scale:

- How fast will your car go?
- How easy is it to use? (Hmm. How do you measure it?)
- How expensive is it to maintain?

Quality Attributes and Architecture

Quality Attributes are system-wide properties

You often need a uniform approach across all components

You usually need to design for the QA from the beginning of the architecture

- At the very least: think about it

But Quality Attributes also tie closely to the code

Early in the architecture

(Performance example)

Agile Mantra:

1. Make it
2. Make it right
3. Make it fast

“Premature optimization is the root of all evil.”

Both the above concentrate on the coding aspect, and ignore the necessity of an architecture that supports necessary performance

If your architecture is inherently slow, all the local optimization won’t save you

(note: does optimization == performance?)

Early in the Architecture

(reliability example)

During coding:

- check the return from every function to make sure it performed properly
- Check transactions to make sure they complete
- Make sure you have try-catch blocks at every point that could fail

At architecture:

- What if a function fails? Devise general approaches
- Decide on transaction rollbacks, voting, etc.

Early in the Architecture

(security example)

Coding:

- Avoid structures that allow buffer overflow
- Implement intrusion detection
- Implement authentication, authorization, encoding, etc.

Architecture:

- Which measures do you use?
- Do you have a security layer in the software?
- An Authentication module? Where does it live?

Early in the Architecture

(security example: Skype)

You are designing Skype. What should you do about security?

Ok, you have password-protected accounts. What if someone forgets their password?

What if the mail to reset their password fails?

Coding: how do you implement that step, given that you are already implementing a trouble ticket system?

Early in the Architecture

(Availability example: Amazon)

Architecture:

- What is the physical architecture of the replicated system?
  - Hot Spare, Warm Spare, Cold Spare, etc.
  - Full duplication vs. N+1 sparing, etc.
  - Already has multiple servers for load, geographic location, etc. Can we use them for availability?
- What are the restart levels, and what do they mean?
  - (How bad does the system have to get before we take a unit out of service?)

Coding:

- Implement "hot spare"
- Implement "escalating restarts"

Early in the Architecture

(Usability example: Parking app for paid parking)

Architecture:

- What is the general flow of the user interactions?
- How to identify a car?
  - Is it based on license places? If so, capture or enter them?

Implementation:

- How should I lay out this screen?
- What font and style?
- What is the flow?

Early in the Architecture

(portability example: Compiler)

Architecture:

- What are the cost and benefit of being completely portable?
- What does portability mean for this system?
- What modules are custom for a particular machine architecture, and which can be shared? (Basic partitioning of the system!)

Coding:

- Make all the code machine-independent
- AND: make it produce different object code for different machines.

Early in the Architecture

(scalability example: Amazon, the early days)

Architecture:

- How to allow for scaling
- What does scaling mean, anyway?
  - Which dimensions?
- Upper limits?

Coding:

- Use coding that allows data to grow, connections to grow, etc.

Early in the Architecture

(maintainability example: Amazon, again)

Architecture:

- Partition the software in ways that are intuitive
- Commonality and Variability analysis
- How can we change one part of the system and integrate it with the rest of the system?
    - And keep the system live during upgrades?

Coding:

- Use good coding practices to keep code clear,

Quality Attribute Decisions

Decisions about quality attributes:

At the architecture level: tend to be strategic

- Includes decisions about tradeoffs among quality attributes

At the implementation level: tend to be tactical

- "Tactics" are implementations of quality attributes.

Footnote: remember definitions of architecture as a set of decisions? These are important decisions!

Where it All Starts

Early: as general requirements and architecture are being formulated

- The two influence each other

For each Quality Attribute:

1. What does it mean in this domain?
   a) Common approach: what ways can this QA fail?
2. How important is it (also relative to others)?
3. What does it do to the architecture?
   a) E.g., does it require certain hardware or software components?
   b) E.g., does it encourage (or discourage) certain patterns?

Determining Importance

How important is a Quality Attribute?

1. Explore the consequences of failure (examples):
    1. Inconvenience
    2. Loss of some time
    3. Loss of some money
    4. Loss of lots of money
    5. Loss of customer trust
    6. Loss of life
2. Consider the likelihood of occurrence
3. Consider difficulty/expense of prevention
4. May consider importance relative to other QAs

(classical risk analysis)

Example: TicketBoss

Quality Attribute	What does it mean?	Acceptable level?	How Important?
Security			
Availability			
Reliability			
Performance			
Scalability			
Usability			
Portability			
Maintainability			
Extensibility			
(fill in others)			

Example: TicketBoss

Quality Attribute	What does it mean?	How Important?
Security	Protect customer data, especially payment info. (no observe) Protect event info. (modification) Roles: customer, admin Protect reservations	Extremely important
Availability	Always can see events, etc. Always can make reservations	6 9's: not important 3 9's: the acceptable threshold
Reliability	Transactions work (rollback) Data integrity Reservations	Very important
Performance	Ticket issuance Response time	
Scalability	How many customers at once How many events (and venues) can we support How many customer profiles? How much history can we have? How many countries do we support?	
Usability	How hard is it to buy a ticket How hard is it to find an event	Very important
Portability	Can we run the application on a different type of server? (maybe)	
Maintainability	How hard is it to maintain the software?	
Extensibility	Events without seats? Restaurant reservations? Airline tickets? Government services? Parking spaces?	
(fill in others)		

Part 2

Selected QAs

Architectural principles for these QAs

- Efficiency
- Complexity
- Scalability
- Adaptability
- Dependability

Efficiency

“Efficiency reflects the system’s ability to meet its performance requirements while minimizing its usage of the resources in its computing environment. A measure of the system’s resource usage economy”

Note: not the same as performance

- Performance is an aspect of efficiency

Performance: the system’s ability to perform required functions within the time needed

- Hard realtime performance
- Soft realtime performance
- Non-realtime performance
- We may not care what resources we consume to do it

However, efficiency can impact performance

Components and Efficiency

Keep components small

- Ideally a component follows the Single Responsibility Principle
- But can reduce a component's reusability (if you care)

Keep Component Interfaces simple and compact

- Follow Interface Segregation Principle
- But not too far: too simple, stripped of context, can mean message processing overhead

Allow multiple interfaces to the same functionality

- It might eliminate need for adapters

Separate processing components from data

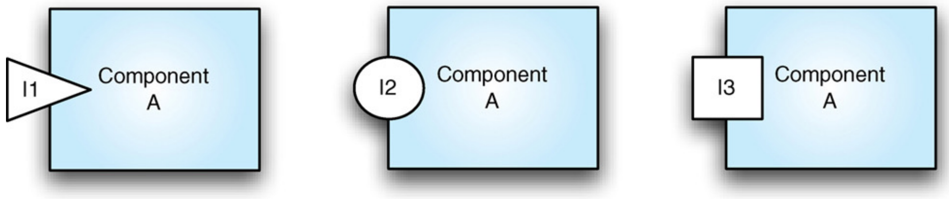
- Can tune either processing or data without affecting the other

Separate data from metadata

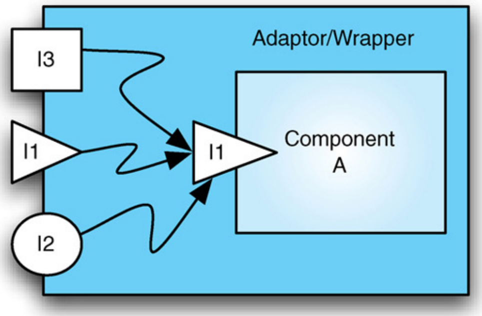
- Data without the metadata is smaller

Exporting multiple interfaces: approaches

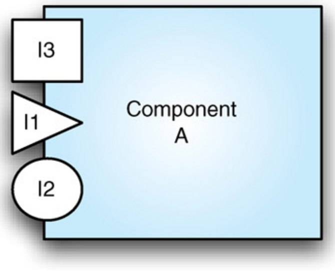
Multiple copies of same component, with different interfaces



A wrapper exposing different interfaces



Component itself has different interfaces



Efficiency and Software Connectors

Carefully select connectors

- Find those that are the best, simplest fit

Use broadcast connectors with caution

- Broadcast may increase flexibility
- But components may receive data they don't need

Make use of asynchronous interaction whenever possible

- So you don't have components waiting
- But there is overhead with context switching

Use location transparency judiciously

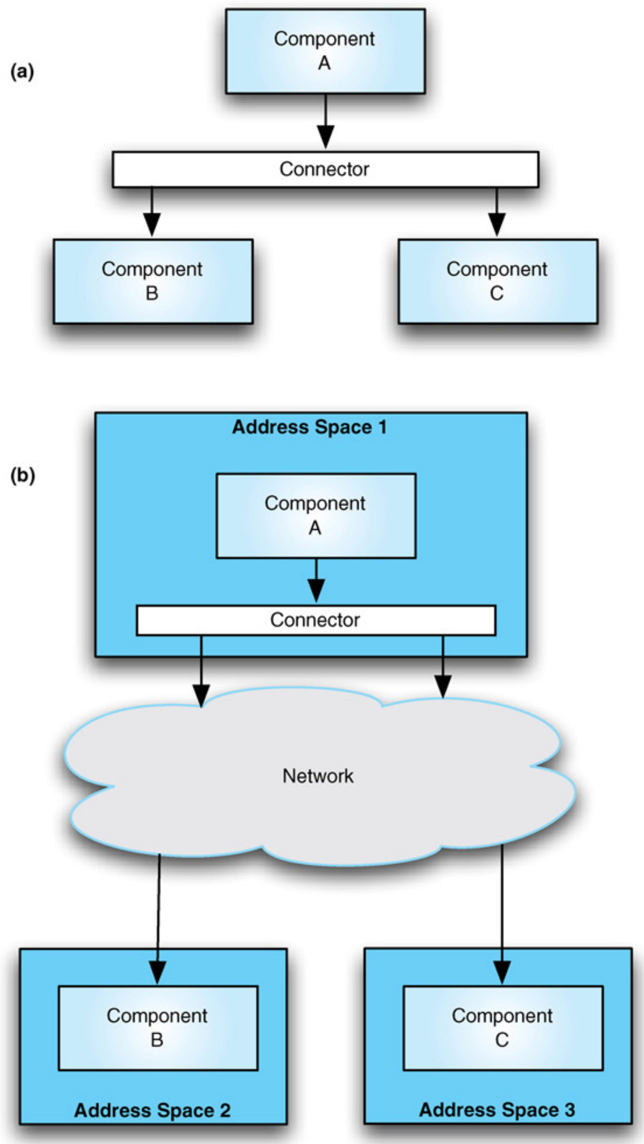
- Distributed components don't have to know their deployment details.
- Flexibility, but may increase overhead
- Among other things, may be difficult to achieve

Location Transparency

Top: simple connection

Bottom: deployed differe

- Looks the same from outs
- But throughput may suffe



Efficiency and Architectural Configurations

Keep Frequently Acting Components Close

- “Close” in what ways? (There are several)
- Reduce number of interactions
- Beware of “pass through” messaging (see picture and System 75 story)

Carefully select and place connectors in the architecture

- Simple specialized usually more efficient than general purpose
- System 75 (again), Call tree system

Efficiency: Memory Usage

Some systems (still) have constrained memory

Example: embedded systems

Systems will be small, so architectures will be simple

Numerous sub-architectural patterns (you could call them tactics) in memory-constrained systems.

Efficiency and Architectural Configurations – Styles and Patterns

Asynchronous architectures (e.g. Publish-Subscribe) don't work well for realtime systems

Large repository systems don't work well for memory constrained systems

Streaming data system MIGHT use event-based architectures, but there is usually a computational penalty

Layers can introduce gratuitous context switches (inefficient)

Incremental data delivery: batch doesn't work, pipes and filters usually doesn't work well

Model-View-Controller not a good fit for distributed/decentralized systems (tight coupling among M, V, & C)

In Client-Server, the server can become a bottleneck

Broker can load balance to maximize efficiency (make sure Broker is efficient, though)

Blackboard: hard to support parallelism

Presentation Abstraction Control: high overhead among agents

Microkernel: can have high overhead

Reflection: Meta-object protocols are often inefficient

Discuss Efficiency and Architecture in General

Discussion point: For efficiency and most other quality attributes, they don’t show up in the architecture as identifiable components

- Possible exceptions: security, availability

Performance

What is the difference between efficiency and performance?

Levels of performance

Levels

- Hard Realtime
- Soft Realtime
- Non-realtime

Aspects

- Latency
- Response time

Discuss architectural implications for each

Discuss what you might trade off for high performance

- Come up with specific examples

Space

Understandability/Maintainability

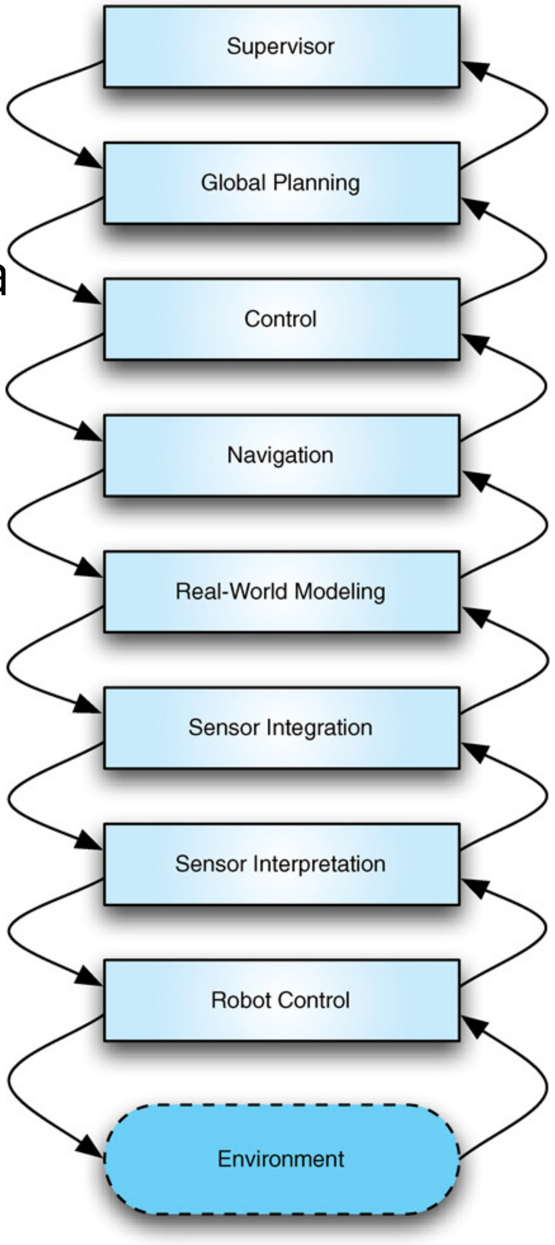
Features

Cost (yes, it’s a quality attribute)

Layers

Mobile robot system

Many layers may slow performa



Performance: Architectural Considerations

Consider network latency

Consider parallel processing

Might use background processes

Reduce context switches

Generally, increased flexibility means decreased performance

Performance optimization can make code harder to understand, impacting maintainability

Performance and Patterns

Layers: not strong

Pipes and Filters: can often optimize for performance.

Broker: Not necessarily good, but can use it for load balancing

Peer to Peer: Be careful of the broadcast overhead

Microkernel, Reflection: Not good

Event-driven: might be good, because you can respond to events quickly (usually).

Complexity

Complexity: definition 1

"Complexity is a system's property that is proportional to the size of the system, the number of its constituent elements, the size and internal structure of each element, and the number and nature of the elements' interdependencies."

- How many parts does it have (relative to its size)?
- How interdependent are the parts?

Why do we care?

Relative to size: (true, but bigger systems are harder to understand anyway.)

Complexity, definition 2:

"Complexity is the degree to which a software system has a design or implementation that is difficult to understand and verify."

- How hard is it to grok?

- Is this always complexity? (Can you have a simple system that is hard to understand?)
- So it's "Understandability"
- But it's really closely tied to complexity (definition1)

Essential complexity

Accidental complexity

(Source: Fred Brooks, “No Silver Bullet”)

Measuring Complexity

Numerous attempts to measure it

- Most based on the code, or the detailed design
  - McCabe Complexity Metric
  - Halstead metrics (a.k.a. “software science”)
- Architecture complexity measures: usually module-based
  - Coupling
  - Fan-in, fan-out
  - Number, size (of each module), etc.
  - Tools used to analyze code and calculate them

An exact number probably doesn’t matter much

We have a pretty good feel for an architecture that is too complex

Discuss architectural implications of complexity (either definition)

How do we design to keep complexity low?

Complexity and Components

Separate concerns into different components

- But don’t go overboard, of course

Keep only the functionality inside the components

- Keep the interaction outside the components

Keep the components cohesive

- Single Responsibility Principle

Be aware of the impact of off-the-shelf components on complexity

- They tend to be general purpose, which means greater complexity

Insulate processing components from changes in data formats

- Related to keeping interaction outside components

Complexity and Connectors

Treat connectors explicitly

- Helps keep interaction out of the components
- Limit different kinds or connectors

Keep only interaction facilities inside connectors

- The functionality goes in the components!

Separate interaction concerns into different connectors

- For example, different connectors for data exchange and for data compression
- Single Responsibility Principle applied to connectors

Restrict interactions facilitated by each connector

- Only involve the components that care about the data you are sending

Be aware of the impact of off-the-shelf connectors on complexity

- Same song, second verse…

Complexity and Architectural Configurations

Eliminate unnecessary dependencies

- It's intuitive that greater interdependencies means greater complexity. Two explicit reasons:
  - There are a greater number of interaction paths
  - It's more difficult to control and predict behavior of the system
- See Linux example, next slides

Manage all dependencies explicitly

- Linux example: note that the documentation of the architecture only mentions some of the dependencies – they are the only ones considered explicitly. (Developers have to discover those dependencies on their own through looking at the code.)

Use hierarchical decomposition

- Linux example: only seven major chunks: easier to grok

Complexity and Information Hiding

Discuss

Off-the-shelf components:

- Can hide information!
- Except when they don’t!

Complexity and selected patterns

Layers: Easy to understand, but beware of “pass-through” messaging, which may increase complexity

Pipes and Filters: Generally pretty simple, if the problem fits it. Note the separation of concerns

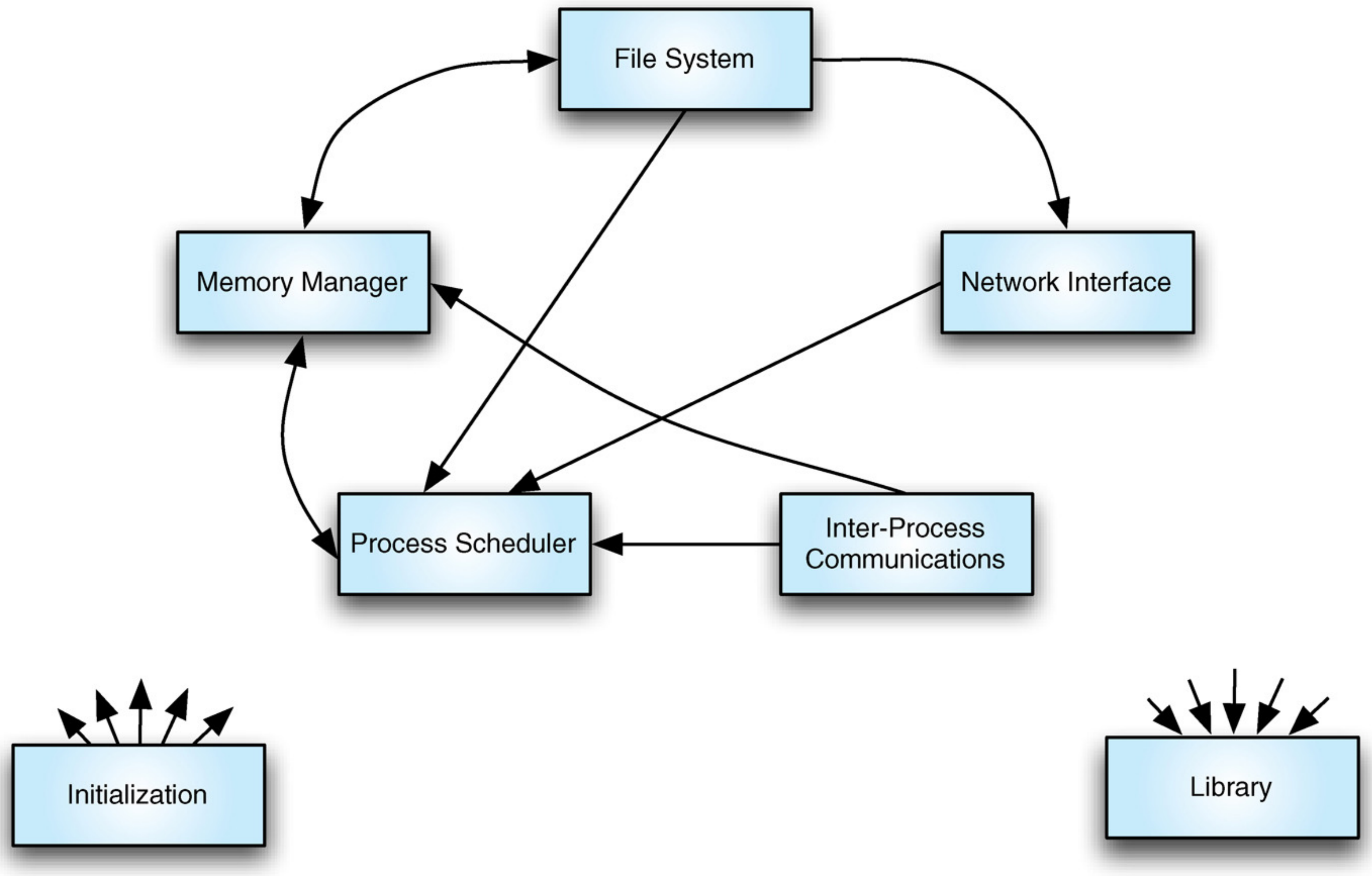
Model View Controller: Only three components, but interaction among them is tight and complex.

Presentation Abstraction Control: good separation of concerns

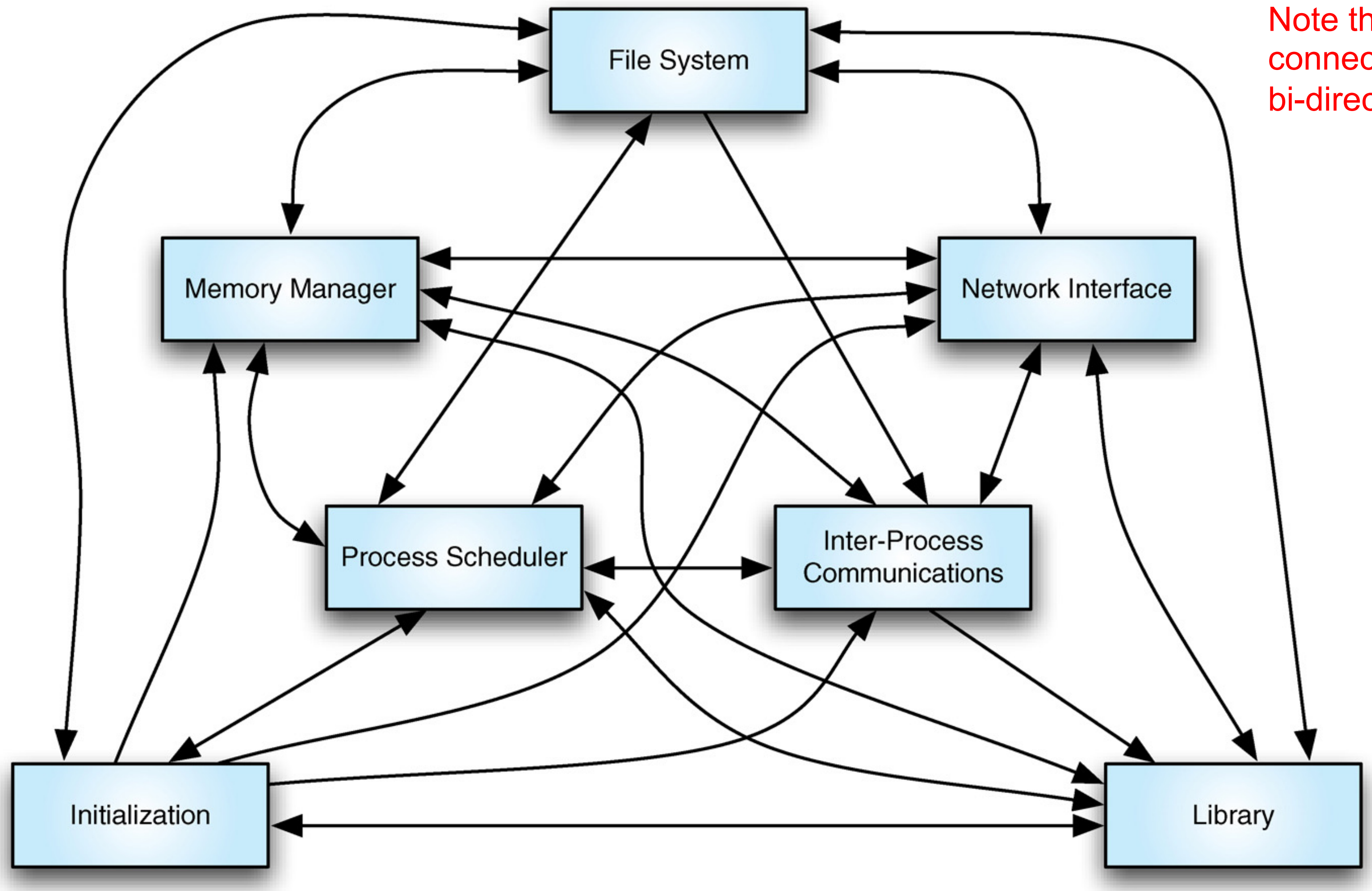
Microkernel: can provide a simpler interface to application, hiding implementation details

Broker: clean separation of concerns, but more components

Linux Architecture, as documented



Linux, as implemented (yikes!)



Note that ALL connectors are bi-directional!

How to Increase Complexity (!)

Speculative Generality

- (But on the other hand: coding unanticipated changes can result in messy code and excess complexity)

Adding Portability (if you don’t need it)

Using Patterns (if you don’t need them)

Adding Features (that you don’t need)

Scalability and Portability

"Scalability is the capability of a software system to be adapted to meet new requirements of size and scope"

- The interesting cases are usually scaling a system *up*
- Can you give examples?

"Portability is a software system's ability to execute on multiple platforms (hardware or software) with minimal modifications and without significant degradation in functional or non-functional characteristics"

- The interesting cases are for heterogeneous platforms, of course

Scalability and Components

Give each component a single, clearly defined purpose

- SRP yet again!
- Allows components to grow or be replicated

Give each component a single, understandable interface

- As above

Do not burden components with interaction responsibilities

- Yet again

Avoid unnecessary heterogeneity

- Keep components compatible with each other

Replicate data when necessary

Scalability and Connectors

Use explicit connectors (e.g., use specified general interfaces)

- Yet again

Give each connector a clearly defined responsibility

- SRP yet again

Choose the simplest connector suited to the task

Be aware of differences between direct and indirect dependencies

- (Law of Demeter)

Do not place application functionality inside connectors

- Yet again

Leverage explicit connectors to support data scalability

Scalability and Architectural Configurations

Avoid System bottlenecks

Try to do things in parallel

- Distribute work across multiple machines
- Because much scaling is increasing the number of requests served, it often works well

Place data sources close to data consumers

Try to make distribution transparent

- Homogeneous distribution is generally better for scalability than heterogeneous distribution

Homogeneous vs. Heterogeneous Distribution

Discuss

Scalability and Patterns

Layers: not particularly strong here

Pipes and Filters: can sometimes extend easily through parallel processing.

Broker: supports scaling very well

Half-Object Plus Protocol

- Supports scalability very well
- Not usually considered an architecture pattern, but it has an architectural flavor.

Peer to Peer: usually scales well

- But careful of the broadcast overhead

Publish Subscribe: good for scalability

Shared Repository: should scale easily

Portability

The Theory:

- Pick a portable language and portable libraries, and you are in good shape!
- AND: Write your code to be portable, and all is well!

The Reality:

- It’s not that easy!
- Recent news:
  - A chemical research library (written in Python) produced different results when run on different OS (about a 1% difference): The implementation of a python library returned a list of files in a different order.

Portability and Patterns

Microkernel/Virtual Machine: That's what it's all about

Layers: very strong

Many patterns have separation of concerns that support portability:

- Client-Server, Broker
- MVC, PAC
- Etc.

Other patterns: should generally be neutral

Maintainability

Supporting factors:

- How easy is it to understand the system (including the code)
- Simplicity (~Complexity)
- Separation of Concerns
- External libraries and other resources used

- A lot of sub-architectural design and coding practices

Note that a lot of maintainability is below the level of architecture

Maintainability and Patterns

Layers: strong: separate modification and testing of layers; supports reuse

Pipes and Filters: can modify filters (somewhat) independently

Blackboard: extendable with independent agents, but difficult to test

Model View Controller: tight coupling impedes maintainability

Presentation Abstraction Control: separation of concerns supports maintainability

Microkernel: easy to extend, separation of concerns

Broker: separation of clients from servers allows separate modification

Dependability

"*Reliability* is the probability that the system will perform its intended functionality under specified design limits, without failure, over a specified time period."

"*Availability* is the probability that the system is operational at a given time."

"A software system is *robust* if it is able to respond adequately to unanticipated run time conditions."

"A software system is *fault-tolerant* if it is able to respond gracefully to failures at run time."

"*Survivability* is a software system's ability to resist, recognize, recover from, and adapt to mission-compromising threats."

"*Safety* denotes the ability of a software system to avoid failures that will result in loss of life, injury, significant damage to property, or destruction of property."

Dependability

(See Deck Module 06 extra)

Security

Some security measures can be added after development but is very helpful to design for security right from the beginning.

This discussion provides an overview of security concepts and how they interact with software architecture

Attacks: One way to look at them

Basic types of attacks on data:

- Access it
- Corrupt (modify) it
- Remove it (special case of modification)
- Prevent access to it (includes removing it)

Basic types of attacks on system resources (e.g., CPU):

- Use it
- Change what someone is running
- Prevent it from running

(Obviously, they all go together)

Three main aspects of Security

Confidentiality

Integrity

Availability

Confidentiality

Preventing unauthorized parties from accessing information

- Or perhaps even being aware of the existence of the information
- Software systems should protect confidential information from being intercepted.
- System should store sensitive data in a secure way so unauthorized users cannot discover content or existence of such data.
- Data aggregation and partial data:
  - Some data may be public in the aggregate, but private individually (example: census data)
  - Some data may be public in a piece, but private when put together
  - Data protection includes protection from deriving data inferentially

Integrity

Only authorized parties can manipulate the data

- And only in authorized ways

Analogous constructs at the programming language level:

- Java: private, package, protected, public access levels
- C++: “const” prevents modification

At software architecture level:

- Watch interfaces

Integrity requires that user identity be established

- User authentication process (e.g. username/password)

Availability

Resources are available if they are accessible by authorized parties on all appropriate occasions

Malicious actions can make resources unavailable (a denial of service (DoS) attack)

Distributed applications may be vulnerable to a distributed denial of service (DDoS) attack.

Design Principles

Least Privilege

- Give each component only the privileges it requires

Fail-save defaults

- Deny access if explicit permission is absent

Economy of mechanism

- Adopt simple security mechanisms; avoid doing the same check in different places

Complete mediation

- All accesses to entities are checked to ensure they are allowed

Open design

- The security of a mechanism should not depend on the secrecy of its design or implementation

Separation of Privilege

- Do not grant permission based on a single condition
- Sensitive operations should require the cooperation of more than one key party

Least common mechanism

- Mechanisms used to access separate resources should not be shared
- The compromise of one resource should not allow the compromise of another

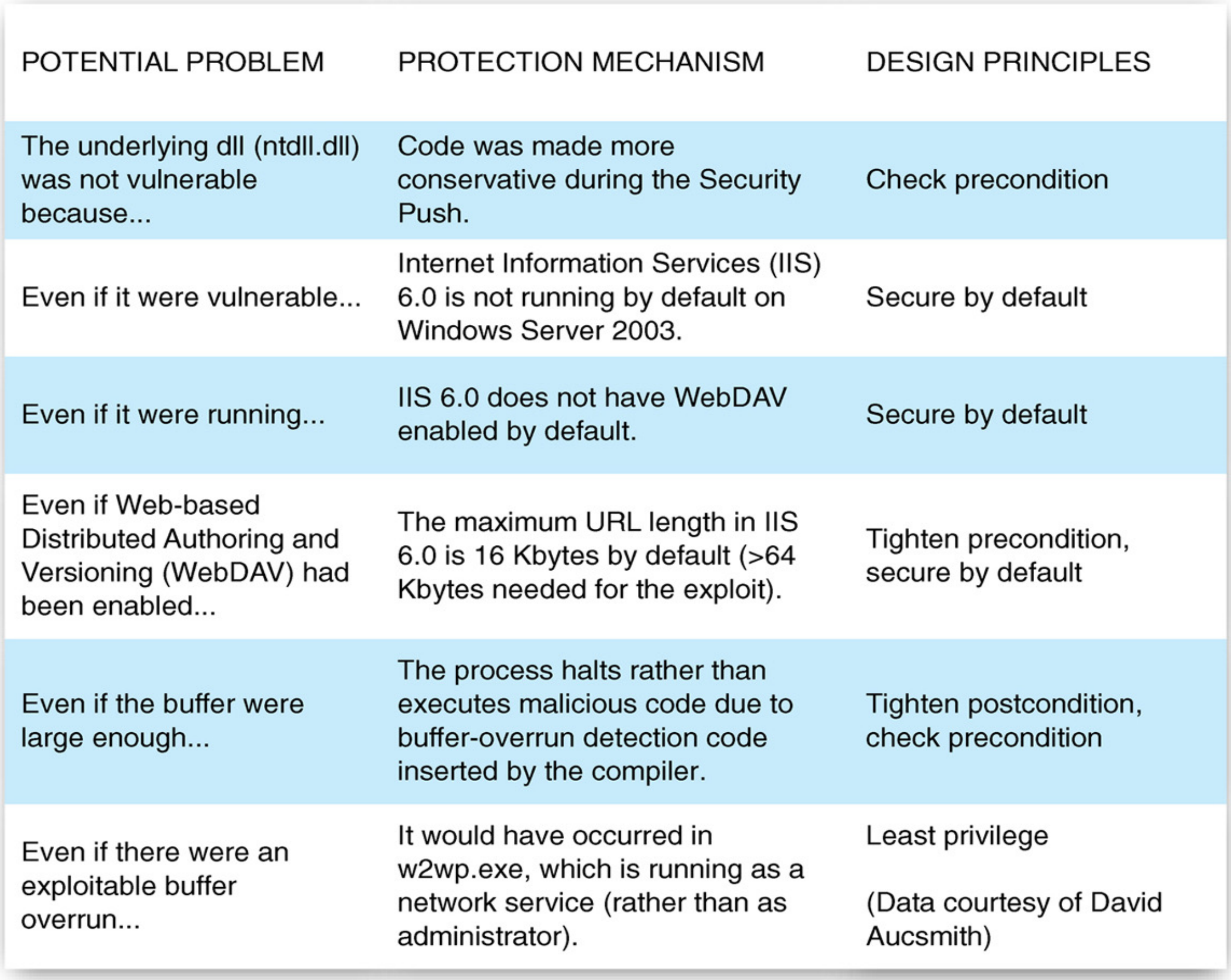
Psychological Acceptability

- Security mechanisms should match the mental model of the users
- Security shouldn’t make it harder to use by legitimate users

Defense in depth

- Have multiple defensive countermeasures

Defense in Depth: Microsoft Internet Information Service



POTENTIAL PROBLEM	PROTECTION MECHANISM	DESIGN PRINCIPLES
The underlying dll (ntdll.dll) was not vulnerable because...	Code was made more conservative during the Security Push.	Check precondition
Even if it were vulnerable...	Internet Information Services (IIS) 6.0 is not running by default on Windows Server 2003.	Secure by default
Even if it were running...	IIS 6.0 does not have WebDAV enabled by default.	Secure by default
Even if Web-based Distributed Authoring and Versioning (WebDAV) had been enabled...	The maximum URL length in IIS 6.0 is 16 Kbytes by default (>64 Kbytes needed for the exploit).	Tighten precondition, secure by default
Even if the buffer were large enough...	The process halts rather than executes malicious code due to buffer-overrun detection code inserted by the compiler.	Tighten postcondition, check precondition
Even if there were an exploitable buffer overrun...	It would have occurred in w2wp.exe, which is running as a network service (rather than as administrator).	Least privilege (Data courtesy of David Aucsmith)

guiar, Neil Harrison

Access Control Models

Classic discretionary access control

- A set of subjects that have privileges (permissions)
- A set of objects on which these privileges can be exercised
- An access matrix specifies the privilege a subject has on a particular object
- Commonly implemented as an access control list

Role-based access control

- Roles become the entities that authorized with permissions
- Roles can form a hierarchy
- Principals are assigned one or more roles

Mandatory access control

- Multilevel security environments
- Each subject and each object are assigned a security label

Connector-Centric Architectural Access Control

Basic concepts

- Subject: the user on whose behalf a piece of software executes
- Principal: a subject can take upon multiple principals – they encapsulate the credentials that a subject possesses to acquire permissions
- Resource: an entity for which access should be protected.
- Permission: the operations on a resource a component may perform
- Privilege: describes what permissions a component possesses.
- Policy: ties together the above concepts: specifies what privileges a subject, with a given set of principals, should have in order to access resources

Role of Architectural Connectors

Architectural access control is centered on connectors because they propagate privileges that are necessary for access control decisions

Supply security contract:

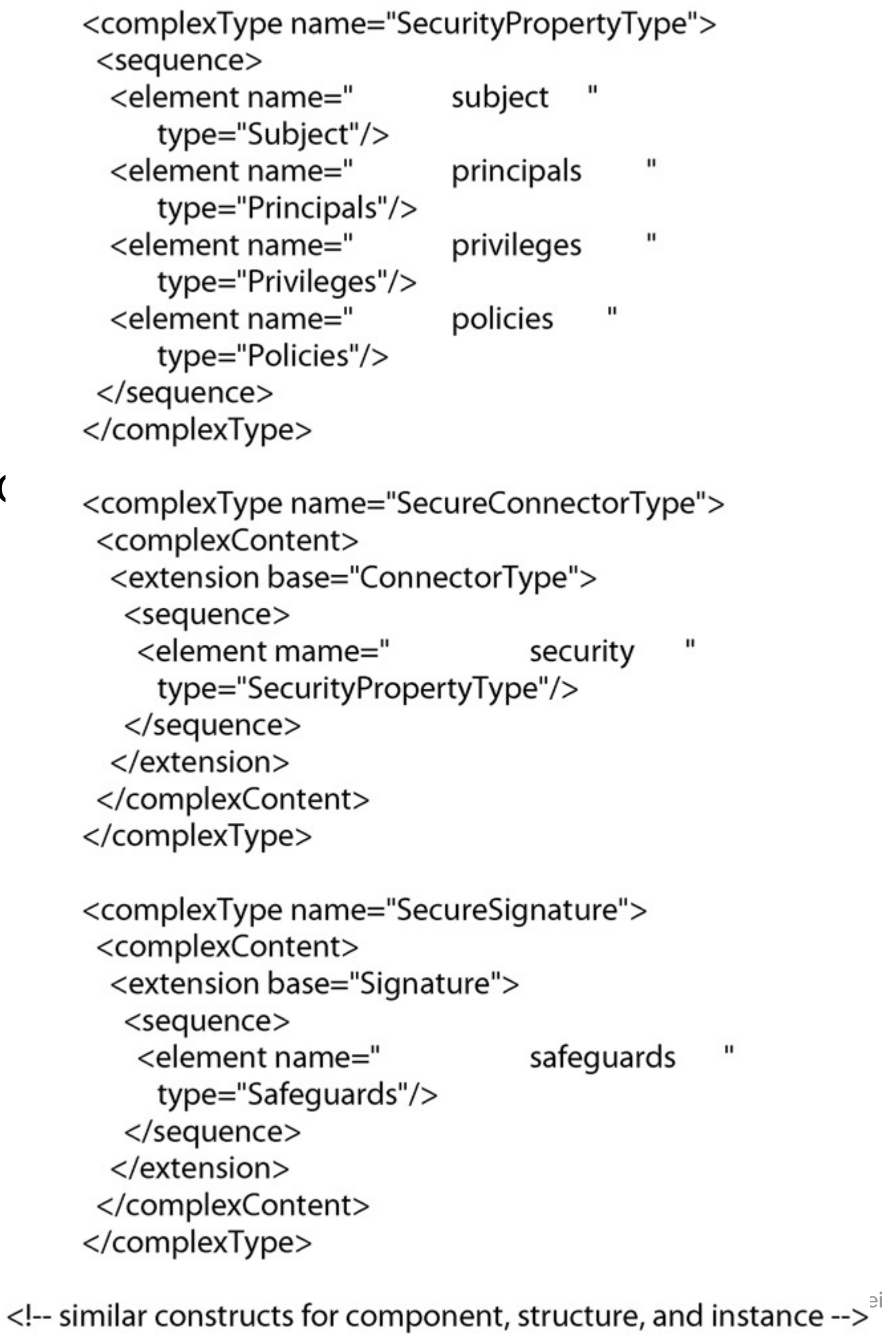
- Specifies the privileges and safeguards of an architectural element

Regulate and enforce contract:

- Can determine the subjects for which the connected components are executing
- Also determine whether components have sufficient privileges
- Might provide secure interaction between insecure components

If model is formally specified, you can write an algorithm to check architectural access contr

formality



```
<complexType name="SecurityPropertyType">
 <sequence>
  <element name="          subject    "
      type="Subject"/>
  <element name="          principals      "
      type="Principals"/>
  <element name="          privileges      "
      type="Privileges"/>
  <element name="          policies      "
      type="Policies"/>
 </sequence>
</complexType>

<complexType name="SecureConnectorType">
 <complexContent>
  <extension base="ConnectorType">
   <sequence>
    <element mame="            security     "
      type="SecurityPropertyType"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>

<complexType name="SecureSignature">
 <complexContent>
  <extension base="Signature">
   <sequence>
    <element name="            safeguards     "
      type="Safeguards"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>

<!-- similar constructs for component, structure, and instance -->
```

Access control check algorithm

```
Input   : an outgoing interface, Accessing,
  and an incoming interface, Accessed

Output  :   grant   if the Accessing can access
  the Accessed,             deny  if the Accessing
  cannot access the Accessed

Begin
   if   (there is no path between Accessing and Accessed)
  return          deny ;
   if   (Accessing and Accessed are connected directly)
  DirectAccessing = Accessing;
   else
  DirectAccessing = the element nearest to Accessed in the path;
 Get AccumulatedPrivileges for
  DirectAccessing from the owning element, the type, the containing
  sub-architecture, the complete architecture, and the
  connected elements;
 Get AccumulatedSafeguards for Accessed from the owning element, the
  type, the containing sub-architecture, and the complete architecture;
 Get AccumulatedPolicy for Accessed from similar sources;
   if   (AccumulatedPolicy exists)
      if   (AccumulatedPolicy grants access)
   return          grant   ;
      else
   return          deny ;
   else
      if   (AccumulatedPrivileges contains AccumulatedSafeguards)
   return          grant   ;
      else
   return          deny ;
End ;
```

Example: Secure Cooperation

Uses the C2 architecture pattern

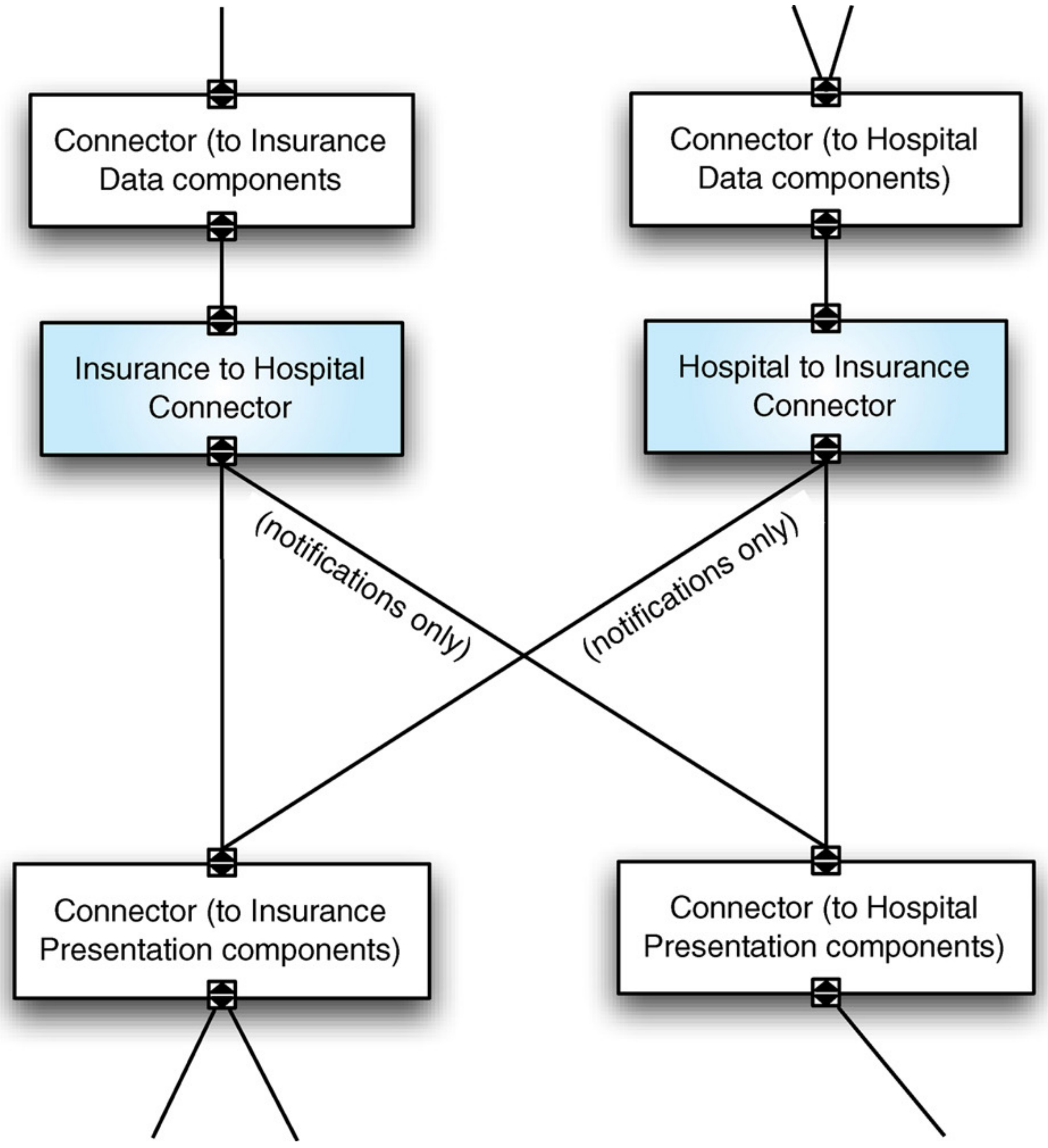
Two parties can share data with each other, but do not necessarily trust each other

- So the data must be subject to control of each party

Two parties: insurance company and hospital

Figure:

- Secure connector on each side


Connector (to Insurance Data components
Connector (to Hospital Data components)
Insurance to Hospital Connector
Hospital to Insurance Connector
(notifications only)
(notifications only)
Connector (to Insurance Presentation components)
Connector (to Hospital Presentation components)

Trust Management

Decentralized applications, where entities do not have complete information about each other

- Thus, must make local decisions autonomously
- Entities must account for the possibility that other entities may be malicious
- Without a central authority that can manage the entities, each must adopt suitable measures to safeguard itself

It is therefore critical to choose an appropriate trust management scheme for a decentralized application

- And implement appropriate protection measures

Trust: “a particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both before he can monitor such an action and in a context in which it affects his own action.”

Reputation

Closely related to trust

“An expectation about an individual’s behavior, based on information about or observations of its past behavior.”

Trust management systems may use reputation to determine the trustworthiness of an entity (reputation-based systems)

- Several common ones
- May be centralized or decentralized

Example: eBay

- Buyers and sellers rate each other after a transaction
- Centralized reputation management system
- System can be manipulated (any system can): in 2000, peers established positive ratings, then used their reputation to start high-priced auctions, received payment, and disappeared.

Threats to Decentralized Systems

Impersonation

Fraudulent Actions

Misrepresentation

Collusion

Denial of service

Addition of unknowns

Deciding whom to trust

Out-of-band knowledge

Measures to Address Threats

Authentication

Separation of Internal Beliefs and Externally Reported Information

Making Trust Relationships Explicit

Comparable Trust

Guidelines to Incorporate into Architecture

Digital Identities

Separation of Internal and External Data

Making Trust Visible

Expression of Trust

Resultant Architectural Style

- Layered architecture works well
- C2 is an event-based layered architecture
- Extention: Practical Architectural style for Composing Egocentric applications (PACE)
  - (one example of an architecture designed to incorporate trust)