SE 577
Software Architecture

Requirements for Architecture

Acknowledgement

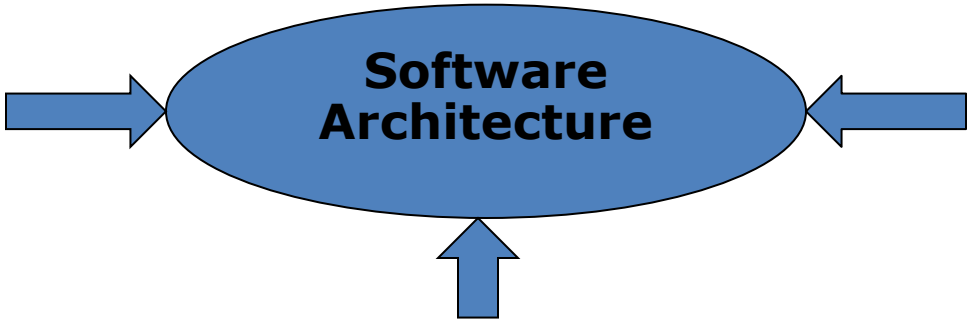
- Material from several sources including:
  - Ian Gorton. *Essential Software Architecture (2nd Edition)*, Springer-Verlag.
  - R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory and Practice*, Wiley.

Requirements

- Solid, concise, on-time requirements are a necessary foundation for a successful architecture.
- We know from experience that most of the critical issues identified during formal project reviews are in the Requirements area.
- What type of requirements shape the architecture?

Requirements That Shape Architecture

Functional Requirements
what the system must do



Quality Attributes
performance, security, modifiability, …

Constraints
decisions made externally (e.g., which OS, which devices, etc.)

Examples

- A typical architecture requirement :
  - "*Communications between components must be guaranteed to succeed with no message loss*"
- Some architecture requirements are constraints:
  - "*The system must use our standard Envoy based API gateway to proxy and secure all web service requests*"
- Constraints impose restrictions on the architecture and are (almost always) non-negotiable.
- They limit the range of design choices an architect can make.

Constraints

Constraint	Architecture Requirement
Business	The technology must run in the Azure cloud, as we want to sell this to Microsoft.
Development	The system must be written in Java so that we can use existing development staff.
Schedule	The first version of this product must be delivered within six months.

Quality Attributes

- QAs are part of an application's Non-Functional Requirements
  - "how" the system achieves its functional requirements
- There are many QAs
- Architect must decide which are important for a given application
  - Understand implications for application
  - Understand competing requirements and trade-offs

Quality Attributes (cont'd)

- Often know as –*ilities*
  - Reliability
  - Availability
  - Portability
  - Scalability
  - Performance (!)

Quality Attribute Requirements

Quality Attribute	Architecture Requirement
Performance	Application performance must provide sub-four second response times for 90% of requests.
Scalability	The application must be able to handle a peak load of 500 concurrent users during the enrollment period.
Modifiability	The architecture must support a phased migration from the current Forth Generation Language (4GL) version to a .NET systems technology solution.
Availability	The system must run 24x7x365, with overall availability of 0.99.
Resource Management	The server component must run on a low end office-based server with 512MB memory.
Usability	The user interface component must run in an Internet browser.
Reliability	No message loss is allowed, and all message delivery outcomes must be known with 30 seconds.
Security	All communications must be authenticated and encrypted using certificates.

Quality Attribute Specification

- Architects are often told:
  - *"My application must be fast/secure/scale"*
- Far too imprecise to be any use at all.

- Quality attributes (QAs) must be made precise/measurable for a given system design, e.g.
  - *"It must be possible to scale the deployment from an initial 100 geographically dispersed user desktops to 10,000 without an increase in effort/cost for installation and configuration."*

Quality Attribute Specification

- QA's must be concrete.
- But what about testable?
  - Test scalability by installing system on 10K desktops?
- Often careful analysis of a proposed solution is all that is possible.
- "It's all talk until the code runs".

Performance

- Many examples of poor performance in enterprise applications.
- Performance requires a:
  - Metric of amount of work performed in unit time
  - Deadline that must be met
- Enterprise applications often have strict performance requirements, e.g.
  - 1000 transactions per second
  - 3 second average latency for a request

Performance: Throughput

- Measure of the amount of work an application must perform in unit time
  - Transactions per second
  - Messages per minute
- Is required throughput:
  - Average?
  - Peak?
- Many system have low average but high peak throughput requirements

Performance: Throughput (cont'd)

- System must achieve 100 mps throughput
  - BAD!!
- System must achieve 100 mps peak throughput for *PaymentReceived* messages
  - GOOD!!!

Performance: Response Time

- Measure of the latency an application exhibits in processing a request
- Usually measured in (milli)seconds
- Often an important metric for users
- Is required response time:
  - Guaranteed?
  - Average?
- E.g. 95% of responses in sub-4 seconds, and all within 10 seconds

Performance: Deadlines

- "something must be completed before some specified time"
  - Payroll system must complete by 2am so that electronic transfers can be sent to bank
  - Weekly accounting run must complete by 6am Monday so that figures are available to management
- Deadlines often associated with batch jobs in IT systems.

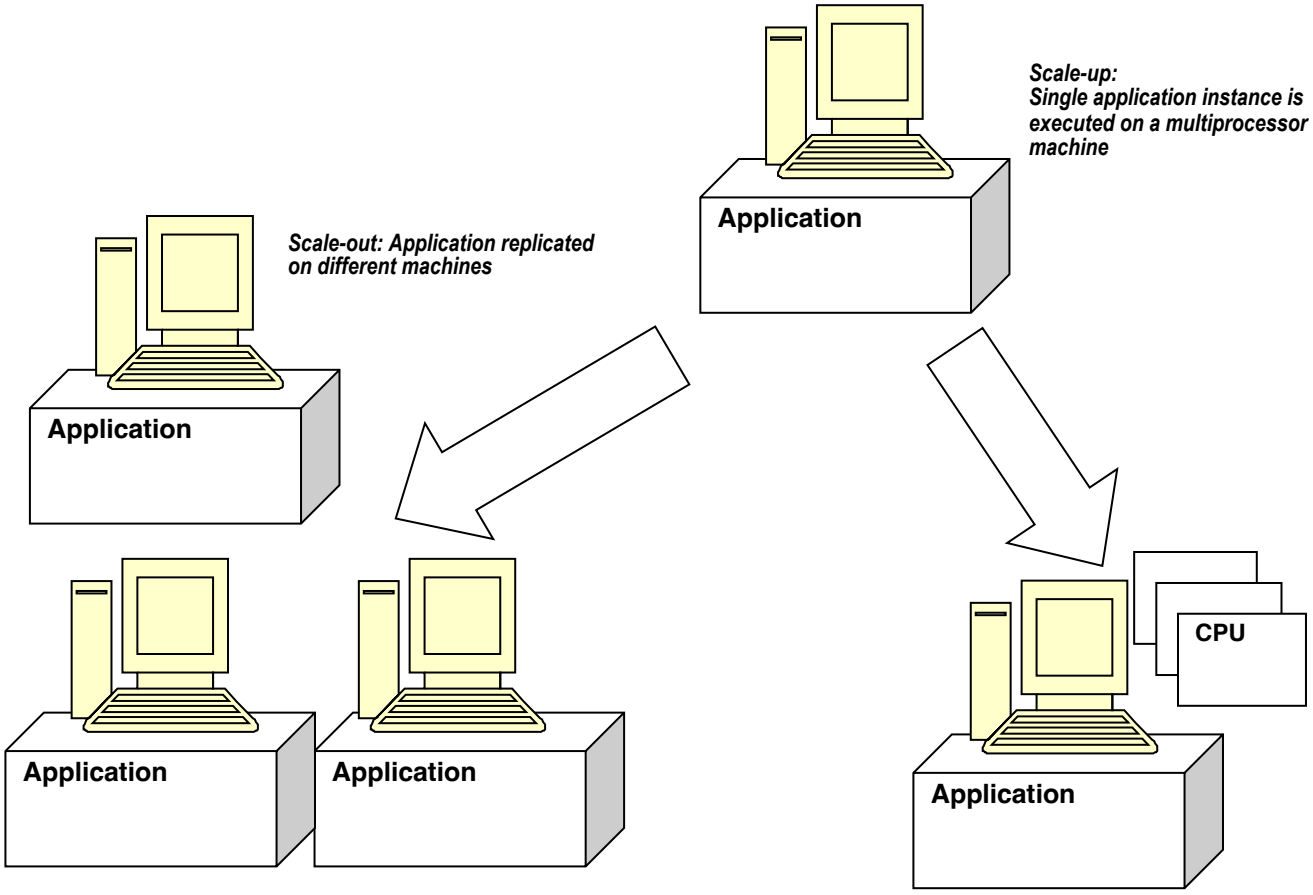
Scalability

- Scalability: How well a solution to some problem will work when the size of the problem increases.
- 4 common scalability issues in IT systems:
  - Request load
  - Connections
  - Data size
  - Deployments

Scalability: Request Load

- How does a 100 tps application behave when simultaneous request load grows?
  - e.g., from 100 to 1000 requests per second?
- Ideal solution, without additional hardware capacity:
  - as the load increases, throughput remains constant (i.e. 100 tps), and response time per request increases only linearly (i.e. from 1 second to 10 seconds).

Scalability – Add more hardware …



Scalability: Connections

- What happens if number of simultaneous connections to an application increases
  - If each connection consumes a resource?
  - Exceed maximum number of connections?
- ISP example:
  - Each user connection spawned a new process
  - Virtual memory on each server exceeded at 2000 users
  - Needed to support 100Ks of users
  - Tech crash ….

Scalability: Data Size

- How does an application behave as the data it processes increases in size?
  - Database table size grows from 1 million to 20 million rows?
  - Image analysis algorithm processes images of 100MB instead of 1MB?
  - Chat application sees average message size double?
- Can application/algorithms scale to handle increased data requirements?

Scalability: Deployment

- How does effort to install/deploy an application increase as installation base grows?
  - Install new users?
  - Install new servers?
- Solutions typically revolve around automatic download/installation
  - E.g. downloading applications from the Internet

Scalability thoughts

- Scalability often overlooked.
  - Major cause of application failure
  - Hard to predict
  - Hard to test/validate
  - Reliance on proven designs and technologies is essential

Modifiability

- Modifications to a software system during its lifetime are a fact of life.
- Modifiable systems are easier to change/evolve
- Modifiability should be assessed in context of how a system is likely to change
  - No need to facilitate changes that are highly unlikely to occur
  - Over-engineering!

Modifiability

- Modifiability measures how easy it **may** be to change an application to cater for new (non-) functional requirements.
  - **'may'** – nearly always impossible to be certain
  - Must estimate cost/effort
- Modifiability measures are only relevant in the context of a given architectural solution.
  - Components
  - Relationships
  - Responsibilities

Modifiability Scenarios

- The COTS speech recognition software vendor goes out of business, and we need to replace this component.
- The application needs to be ported from Linux to the Microsoft Windows platform.
- Provide access to the application through firewalls in addition to existing “behind the firewall” access.

Modifiability Analysis

- Impact is rarely easy to quantify
- The best possible is a:
  - Convincing impact analysis of changes needed
  - A demonstration of how the solution can accommodate the modification without change.
- Minimizing dependencies increases modifiability
  - Changes isolated to single components likely to be less expensive than those that cause ripple effects across the architecture.

Availability

- Key requirement for most IT applications
- Measured by the proportion of the time the system is functional. E.g.
  - 100% available during business hours
  - No more than 2 hours scheduled downtime per week
  - 24x7x52 (100% availability)
- Related to an application's reliability
- Reliability: The probability that a system is functional for a given interval of time.
- Unreliable applications suffer poor availability

Specifying Availability

- The components that determine availability are MTBF (Mean Time Between Failure) and MTTR (Mean Time to Restore).
- Examples:
  - The processor averages 1 outage each month.
  - So, the MTBF = 1 month (or 30 days or 720 hrs, etc.)
  - It takes 2 hours on the average to restore service when the processor goes down.
  - So, the MTTR = 2 hours.
- Availability = (MTBF – MTTR) / MTBF
  = ((720 hrs/month) – 2 hrs) / 720hrs/month
  = 99.72%

Availability

- Strategies for high availability:
  - Eliminate single points of failure
  - Replication and failover
  - Automatic detection and restart
- Recoverability (e.g. a database)
  - The capability to reestablish performance levels and recover affected data after an application or system failure

Security

- Difficult, specialized quality attribute:
  - Lots of technology available
  - Requires deep knowledge of approaches and solutions
- Security is a multi-faceted quality …

Security

- Authentication: Applications can verify the identity of their users and other applications with which they communicate.
- Authorization: Authenticated users and applications have defined access rights to the resources of the system.
- Encryption: The messages sent to/from the application are encrypted.
- Integrity: This ensures the contents of a message are not altered in transit.
- Non-repudiation: The sender of a message has proof of delivery and the receiver is assured of the sender's identity. This means neither can subsequently refute their participation in the message exchange.

Security Approaches

- SSL (Secure Sockets Layer)
- PKI (Public Key Infrastructure)
- Web Services security – JWT / oAuth
- JAAS (Java Authentication and Authorization Service)
- Operating system security
- Database security
- Etc., etc.

Misc. Quality Attributes

- Portability
  - Can an application be easily executed on a different software/hardware platform to the one it has been developed for?
- Testability
  - How easy or difficult is to test an application?
- Supportability
  - How easy an application is to support once it is deployed?

Priorities

- All requirements are not equal
  - **High**: the application must support this requirement.
  - **Medium**: this requirement will need to be supported at some stage
  - **Low**: this is part of the requirements wish list.

Design Trade-offs

- QAs are rarely orthogonal – they interact, affect each other
  - High performance application may be tied to a given platform, and hence not be easily portable.
  - Highly secure system may be difficult to integrate.
  - Highly available application may trade-off lower performance for greater availability.
- Architects must create solutions that makes sensible design compromises
  - Not possible to fully satisfy all competing requirements
  - Must satisfy all stakeholder needs
  - This is the difficult bit!

Define the Criteria for Success

- To know whether the system achieves its *objectives* you must know ***what*** these objectives are and be able to **quantify** them.
  - Project the *expected results*, both positive and negative, of having the system operational (these are used as starting point for the customer acceptance criteria.)
  - Quantify the expected results either relative to the baseline requirements or in absolute terms.
  - Describe what results are to be measured and focus on how they will be used. (Do not describe the implementation of the measurement system, i.e., how the results are to be measured.)
  - Be sure the customer agrees with you.

Example

Let’s consider the following high-level request by the customer:

Develop a system that:

- Remotely controls video cameras
- Supports video processing
- Communicates with other sensors such as radars

Sample System Functionality

- **Camera Control**
  - A GUI must allow for full function camera control (e.g., move the camera up, down, left, right, zoom in and out, etc.)
  - The end-user should be able to configure each camera (e.g., set brightness, contrast, etc.)
- **Video Processing**
  - Video should be displayed in the GUI.
  - The user shall have the ability to annotate the video.
  - All incoming video, and associated annotations made by the end-user, must be recorded for later retrieval and playback.
- **Process Radar Hits**
  - The user should be able to configure and start/stop the operation of the XYZ radar.
  - When the radar detects an object, it should notify the system. Upon receiving the notification, the system should display a special symbol on the user interface to indicate the location of the identified object.

Functionality: Camera Control

- Basic architectural questions related to **cameral control**:
  - How many video cameras should be supported simultaneously?
  - What is an acceptable delay in camera response?
  - What is an acceptable level of quality for the incoming video?

Functionality: Video Processing

- Basic architectural questions related to **video processing**:
  - How much video do we store?
  - How much space does 1 minutes of video take?
  - How do we store the video?
  - What happens when the storage gets filled up?
  - How will the stored video be used?

Functionality: Process Radar Hits

- Basic architectural questions related to **processing input from the radar**:
  - How many radars should we be able to support simultaneously?
  - How many hits per second should we expect from each radar?
  - Do we need to store the incoming hits?
  - In what format should the incoming hits be stored?
  - What should the user be able to do with this information?

Performance Budgets

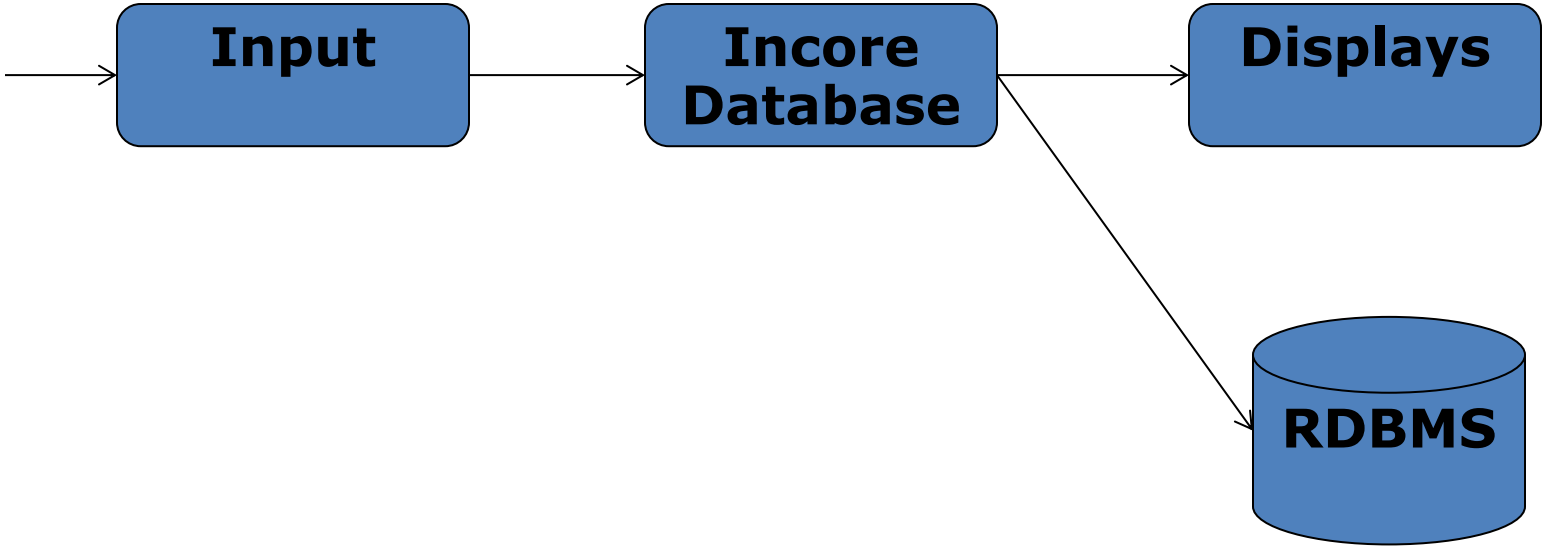
- When designing a system to meet certain levels of performance, it is highly recommended that use the requirements to develop performance budgets.
- Performance budgets should be used during the design of the system to ensure that each subsystem operates within the assumed performance envelope.

Performance Budgeting: An Example

- Consider the following system:
  - There are 20,000 input events per hour into the system.
  - These must be stored in a database for later analysis.
  - Processing is done to categorize these events by type.
  - 10 displays are updated every 15 seconds showing events per type.

High-Level Architecture

- Let's assume that the *proposed* architecture to solve this problem consists of 4 major functional subsystems:



High-Level Architecture (cont'd)

- The design states that:
  - Input 20,000 events per hour.
  - Each event will be categorized and stored in an in-memory database. (This was a design choice to hold data for processing.)
  - Ten displays will be updated every 15 seconds from the in-memory database. The processing is done as part of this function.
  - Every 15 minutes the in-memory database will be written to a relational database on disk for permanent storage.

What next?

- Complete the lower-level design, implement the system, and then test it to ensure that it can handle dealing with the 20,000 events per hour (including processing and displaying).
- There are obvious risks with such approach…
  - What if during testing (that is, after all the work has been completed) we find out the system can't process the 20,000 events per hour?
  - We might need to go back to the drawing board – kind of late in the process ☹
  - Obviously, we better pray that the 2nd time around we have better luck/results.

Another approach…

- As part of the design, go through an exercise to figure out the work (in terms of processing) that each component has to do in order to meet the overall objective.
- If during implementation it is determined that a component is substantially over its “budget”, then design trade offs can be considered.
- What are the benefits of this approach?

The Budgeting Exercise

- How do we go about this ???

- Step 1: Figure out our performance "targets"
  - How much time do we have available for 1 event ???.

The Budgeting Exercise

- How do we go about this ???

- Step 1: Figure out our performance "targets"
  - How much time do we have available for 1 event ???.

    If we are to handle 20,000 events/hr, then the time we have available for 1 event is 180ms.

The Budgeting Exercise

- How do we go about this ???

- Step 1: Figure out our performance "targets"
  - How much time do we have available for 1 event ???.

    If we are to handle 20,000 events/hr, then the time we have available for 1 event is 180ms.

  - How much time do we have available to update each display???

The Budgeting Exercise

- How do we go about this ???

- Step 1: Figure out our performance "targets"
  - How much time do we have available for 1 event ???.

    If we are to handle 20,000 events/hr, then the time we have available for 1 event is 180ms.

  - How much time do we have available to update each display???

    If we are to update 10 displays every 15 seconds, then we have 1.5 seconds to update each display.

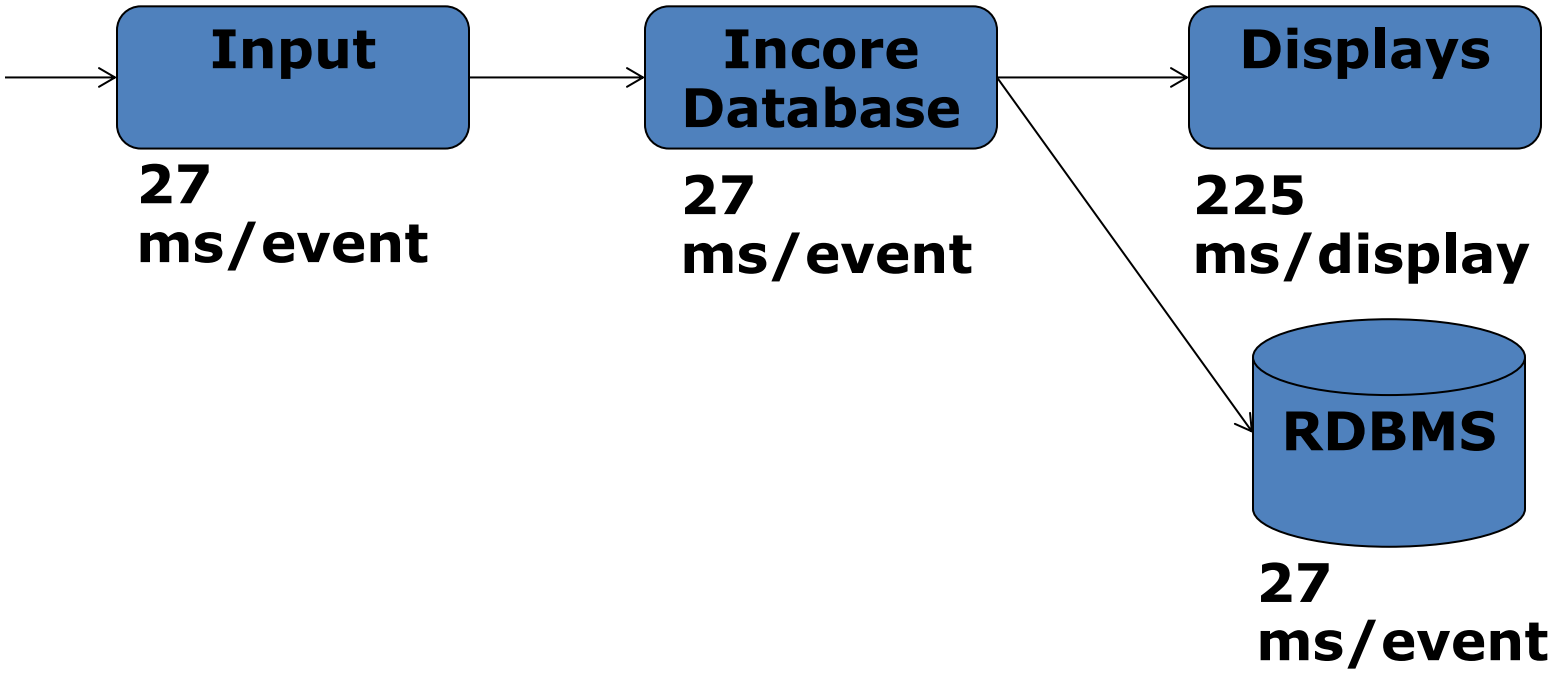
The Budgeting Exercise

- Step 2: Establish budgets
  - Allocate a percent of the work to each component.
    - Why a percent for each component?
  - Very dangerous to allocate 25% to each component – why?
  - Start with allocation of 15% for each of 4 subsystems = 60% total.

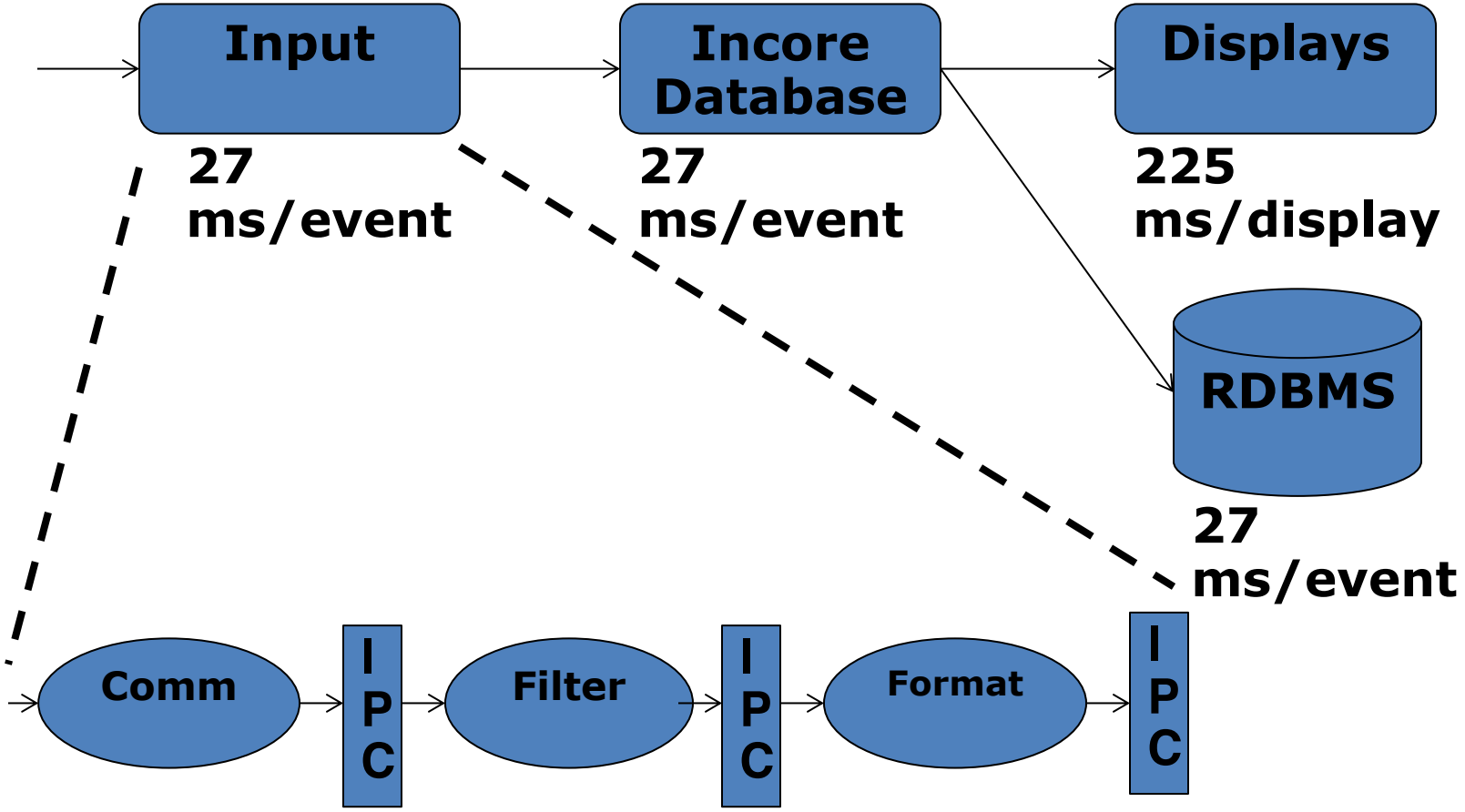
Budgets

- Input:
  - 20,000 events/hr = 180 ms/event
    = 27 ms/event (@ 15% allocated)
- In-core Database:
  - 20,000 events/hr (given storage of all events)
    = 27 ms/event (@ 15% allocated)
- DISPLAYS:
  - 10 Displays every 15 secs = 1.5 secs / display
    = 225 ms/display (@15%)
- RDBMS
  - Also, 27ms/event but batched every 15 minutes.
    (Note: Peaking effect could cause temporary overload or delay of basic collection.)

Budget Summary



Further Budget Allocation



Benefits From Having Budgets

- If during the development, the designer of the DB module determines that 50ms was required to process each input, several design tradeoffs might be considered:

Possible Tradeoffs

- If during the development, the designer of the DB module determines that 50ms was required to process each input, several design tradeoffs might be considered:
  - Reallocate the CPU budget so that the DB gets more of the resources (that is, take away from other modules).
  - Redesign the DB algorithm.
  - Upgrade to a faster CPU that would bring the CPU numbers back in the range of the budget.

Summary

- This is the iterative nature of this process.
- This level of awareness of the resource needs at this low level of design is key to helping determine the feasibility of the system.