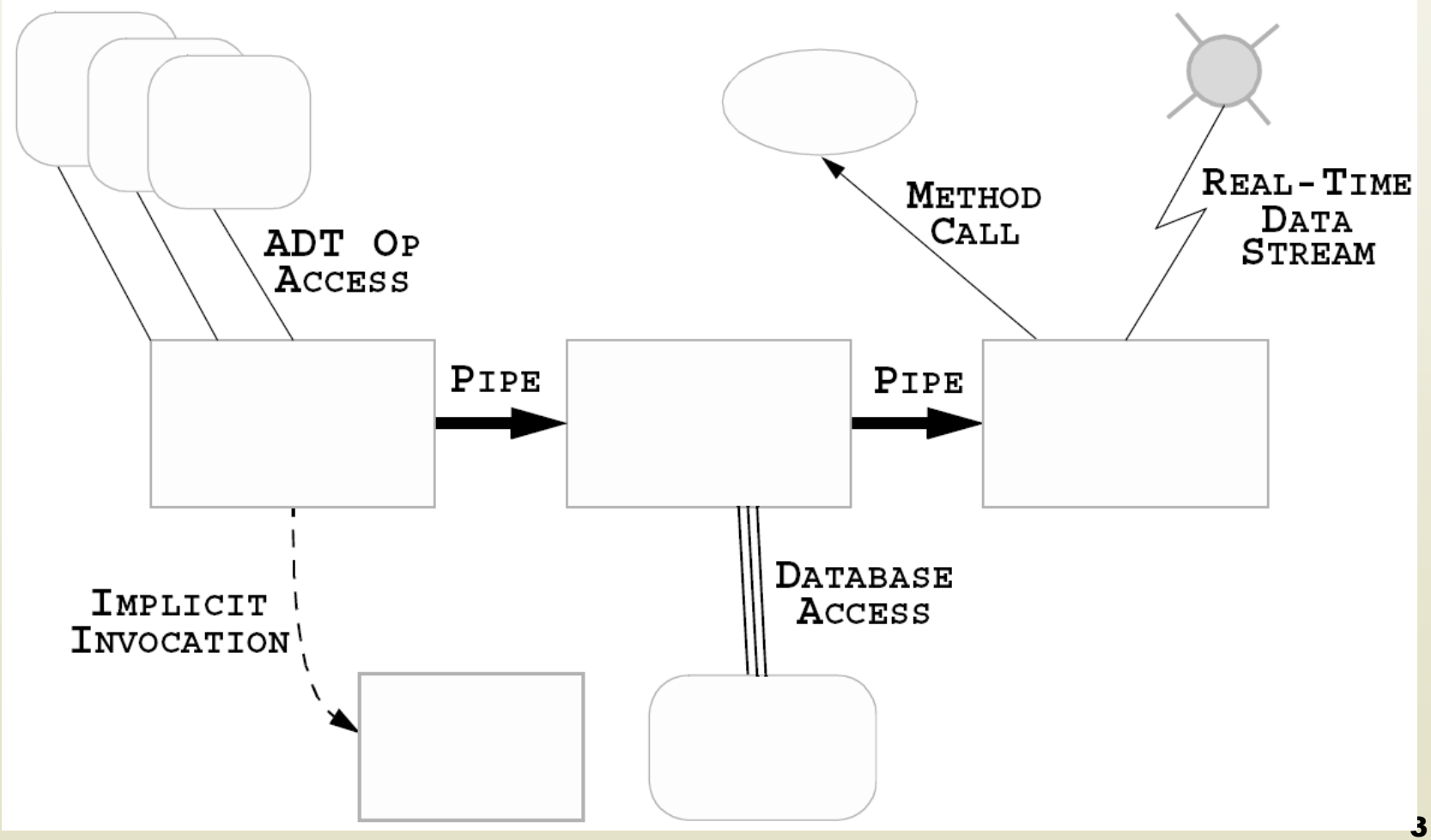
Software Connectors

Software Architecture
Chapter 5

What is a Software Connector?

- Architectural element that models
  - Interactions among components
  - Rules that govern those interactions
- Simple interactions
  - Procedure calls
  - Shared variable access
- Complex & semantically rich interactions
  - Client-server protocols
  - Database access protocols
  - Asynchronous event multicast
- Each connector provides
  - Interaction duct(s)
  - Transfer of control and/or data

Where are Connectors in Software Systems?



Implemented vs. Conceptual Connectors

- Connectors in software system implementations
  - Frequently no dedicated code
  - Frequently no identity
  - Typically, do not correspond to compilation units
  - Distributed implementation
    - Across multiple modules
    - Across interaction mechanisms

Implemented vs. Conceptual Connectors (cont'd)

- Connectors in software architectures
  - First-class entities
  - Have identity
  - Describe all system interaction
  - Entitled to their own specifications & abstractions

Reasons for Treating Connectors Independently

- Connector ≠ Component
  - Components provide application-specific functionality
  - Connectors provide application-independent interaction mechanisms
- Interaction abstraction and/or parameterization
- Specification of complex interactions
  - Binary vs. N-ary
  - Asymmetric vs. Symmetric
  - Interaction protocols

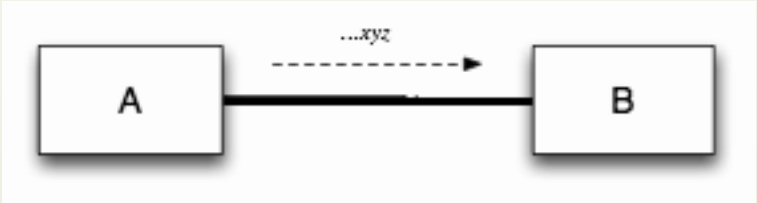
Reasons for Treating Connectors Independently (cont'd)

- Localization of interaction definition
- Extra-component system (interaction) information
- Component independence
- Component interaction flexibility

Benefits of First-Class Connectors

- Separate computation from interaction
- Minimize component interdependencies
- Support software evolution
  - At component-, connector-, & system-level
- Potential for supporting dynamism
- Facilitate heterogeneity
- Become points of distribution
- Aid system analysis & testing

An Example of Explicit Connectors



An Example of Explicit Connectors (cont'd)

- A simple pipe-and-filter architecture consisting of two filters, `A` and `B`, communicate via untyped data streams through the unidirectional pipe connector `P`
- This pipe allows interaction via unformatted streams, consisting of a single interaction channel or 'duct', facilitating unidirectional data transfer, and with cardinality 1, i.e., single sender and single receiver
- Also, the components don't know of each other and there is no buffering of data: if `A` sends the data at a time that `B` cannot receive it, the data is lost

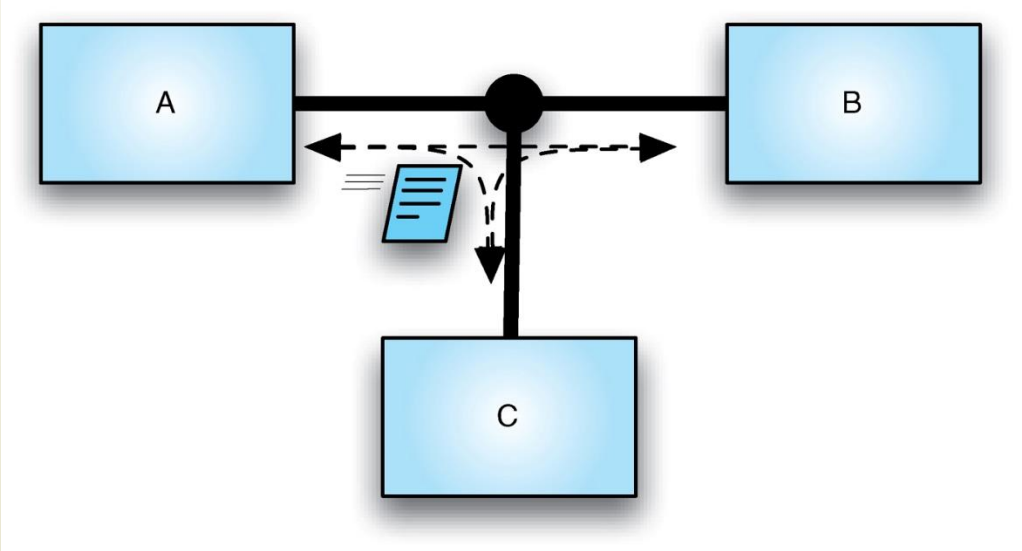
An Example of Explicit Connectors (cont'd)

- Let's assume now that `B` also wants to send information to `A` (e.g., ack. of having received data from `A`)
- Also, we want to ensure delivery of data: if `B` cannot receive the data sent by `A`, the pipe retries until the data is received
- For the first change, a second pipe would be needed from `B` to `A`
- For the second change, we could introduce buffering
- More modifications (e.g., adding more components) would require further addition or replacement of pipes, which would result in the system being down for a considerable amount of time

An Example of Explicit Connectors (cont'd)

- A better solution is to change the nature of the data from unformatted byte streams to discrete, typed packets that can be processed more efficiently
- However, for this solution, pipes don't suffice and what is needed is an event bus, which is able to create ducts between interacting components on the fly, and allows one sender to communicate with multiple observers
- Event buses allow components to be added or removed, and to subscribe to receive certain events, at any time during a system's execution

An Example of Explicit Connectors (cont'd)



An event-based architecture, where components `A` and `B` communicate via typed discrete data packets (events) through an event bus connector, which allows on the fly system modification, e.g., the addition of a new component `C`

Connector Foundations

- The building blocks of every connector are the primitives for managing in a system
  - The *flow of control* (changing the processor program counter)
  - The *flow of data* (performing memory access)
- Also, every connector maintains one or more *channels* (‘*ducts*’) that link the interacting components and support the above-mentioned flows

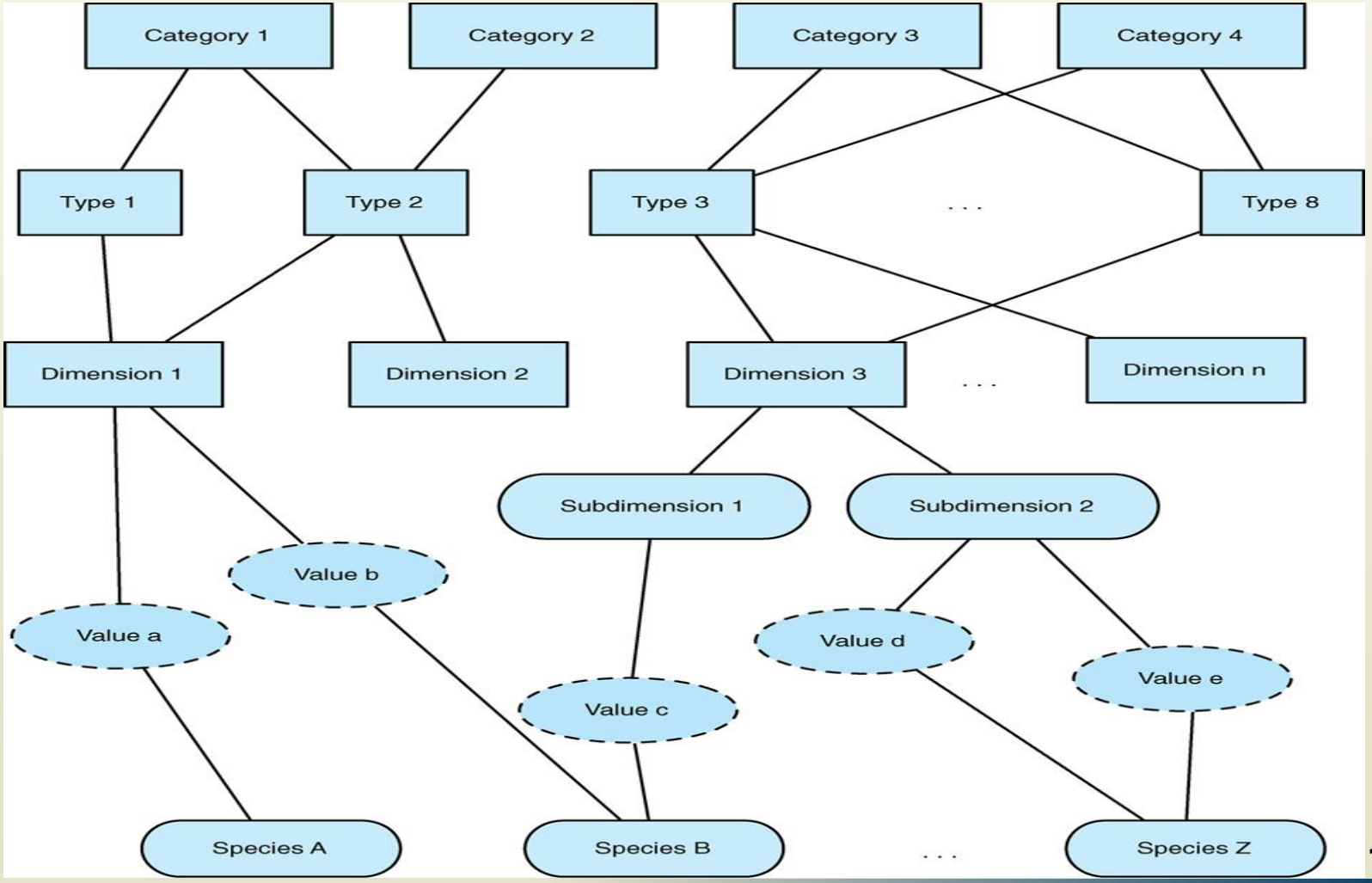
Simple Connectors

- They provide their service simply by forming ducts between components (e.g., module linkers)
- They are typically implemented in programming languages
- They typically provide only one type of interaction service

Complex (or very Complex) Connectors

- They augment ducts with some combination of data and control flow to provide richer interaction services
- Can also have an internal architecture that includes computation and information storage
  - E.g., a load balancing connector would switch incoming traffic based on the current/past load state of components
- Composite connectors are achieved through composition of several connectors (and possibly components), and provided for use as libraries or frameworks
- It is important to be able to reason about their underlying, low-level interaction mechanisms

A Framework for Studying Software Connectors



A Connector Classification Framework

- Each connector is identified by its primary service category
- The characteristics most commonly observed among connectors are positioned towards the top of the framework
- Variations are located towards the bottom
- The framework comprises service categories, connector types, dimensions (and possibly their subdimensions), and values for the dimensions
- A service category represents the broad interaction role the connector fulfils

A Connector Classification Framework (cont'd)

- Connector types discriminate among connectors based on the way in which the interaction services are realized
- Dimensions (and possibly their subdimensions) capture the architecturally relevant details of each connector
- A set of values is associated with a dimension (or subdimension)
- A connector instance can take a number of values from different types

Software Connector Roles

- Locus of interaction among set of components
- Protocol specification (sometimes implicit) that defines its properties
  - Types of interfaces it is able to mediate
  - Assurances about interaction properties
  - Rules about interaction ordering
  - Interaction commitments (e.g., performance)
- Roles ('categories' in the previous classification)
  - Communication
  - Coordination
  - Conversion
  - Facilitation

Connectors as Communicators

- Main role associated with connectors and supporting transmission of data
- Supports
  - Different communication mechanisms
    - E.g., procedure call, RPC, shared data access, message passing
  - Constraints on communication structure/direction
    - E.g., pipes
  - Constraints on quality of service
    - E.g., persistence
- Separates communication from computation
- May influence non-functional system characteristics
  - E.g., performance, scalability, security

Connectors as Coordinators

- Support transfer of control among components
- Components interact by passing the thread of execution to each other
- Control delivery of data
- Separates control from computation
- Orthogonal to communication, conversion, and facilitation
  - Elements of control are in communication, conversion and facilitation
- E.g., function calls and method invocations or signals and load balancing connectors

Connectors as Converters

- Enable interaction of independently developed, heterogeneous or mismatched components
- Mismatches caused by incompatible assumptions about
  - Type
  - Number
  - Frequency
  - Order of interaction among components
- Examples of converters
  - Adaptors for different data formats
  - Wrappers for legacy components

Connectors as Facilitators

- Enable interaction of components intended to interoperate
  - Mediate and streamline interaction
- Govern access to shared information
- Ensure proper performance profiles
  - E.g., load balancing
- Provide synchronization mechanisms
  - Critical sections
  - Monitors

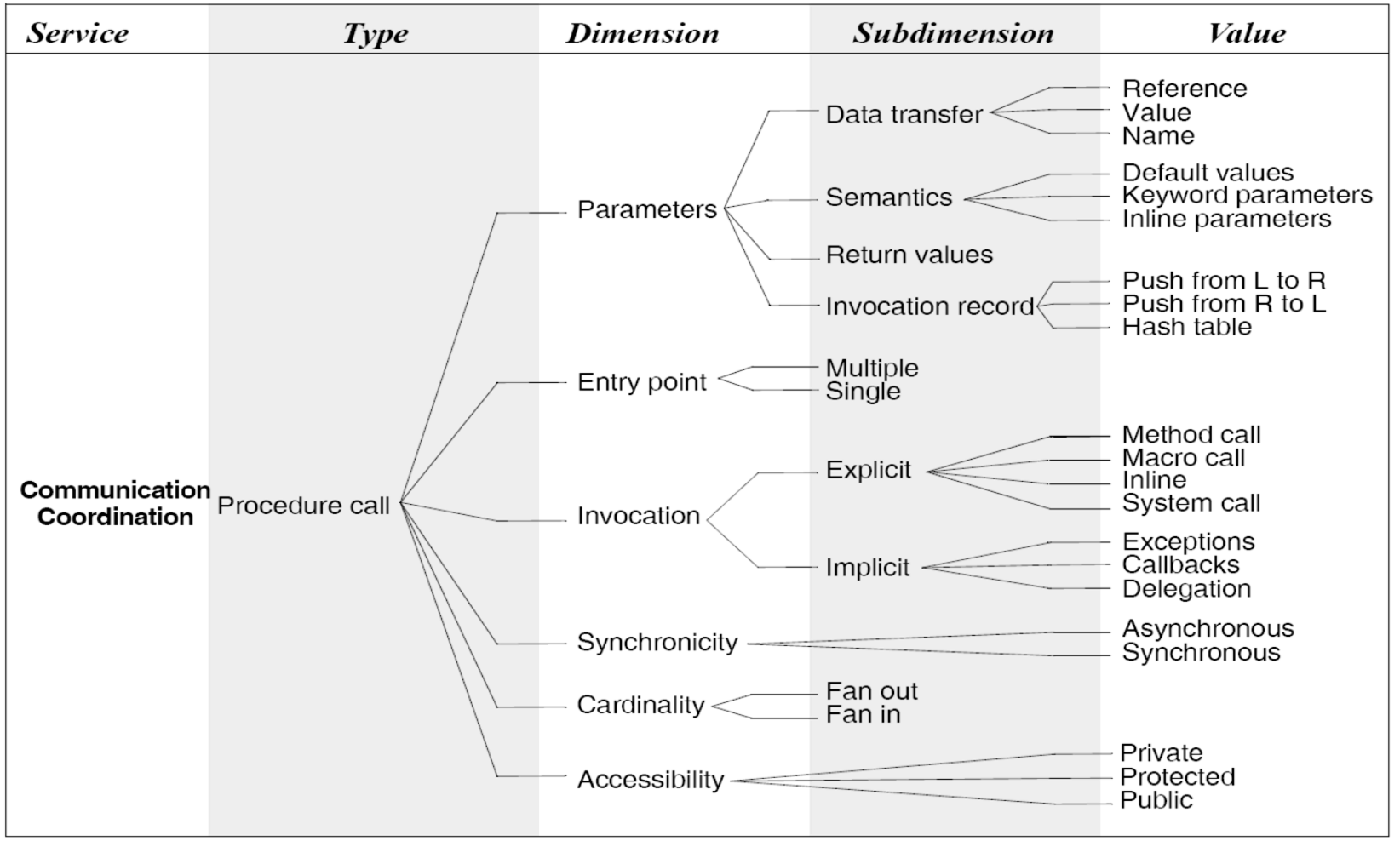
Connector Types and their Variation Dimensions

- Connectors can be further classified based on the way in which they realize interaction services
  - Procedure call
  - Event
  - Data Access
  - Linkage
  - Stream
  - Arbitrator
  - Adaptor
  - Distributor

Procedure Call Connectors

- They are coordination connectors that model the flow of control among components
- They are also communication connectors, transferring data among interacting components by means of parameters and return values
- Examples are OO methods, `fork` and `exec` in Unix, RPC
- The space of options available to a software engineer is shown in the next diagram. E.g., a procedure (method) call in Java implements data transfer of parameters by reference, it may have a return value, it will have a single entry point at the start of the invoked method, it will be a result of explicit invocation, etc.

Procedure Call Connectors Type and Variations



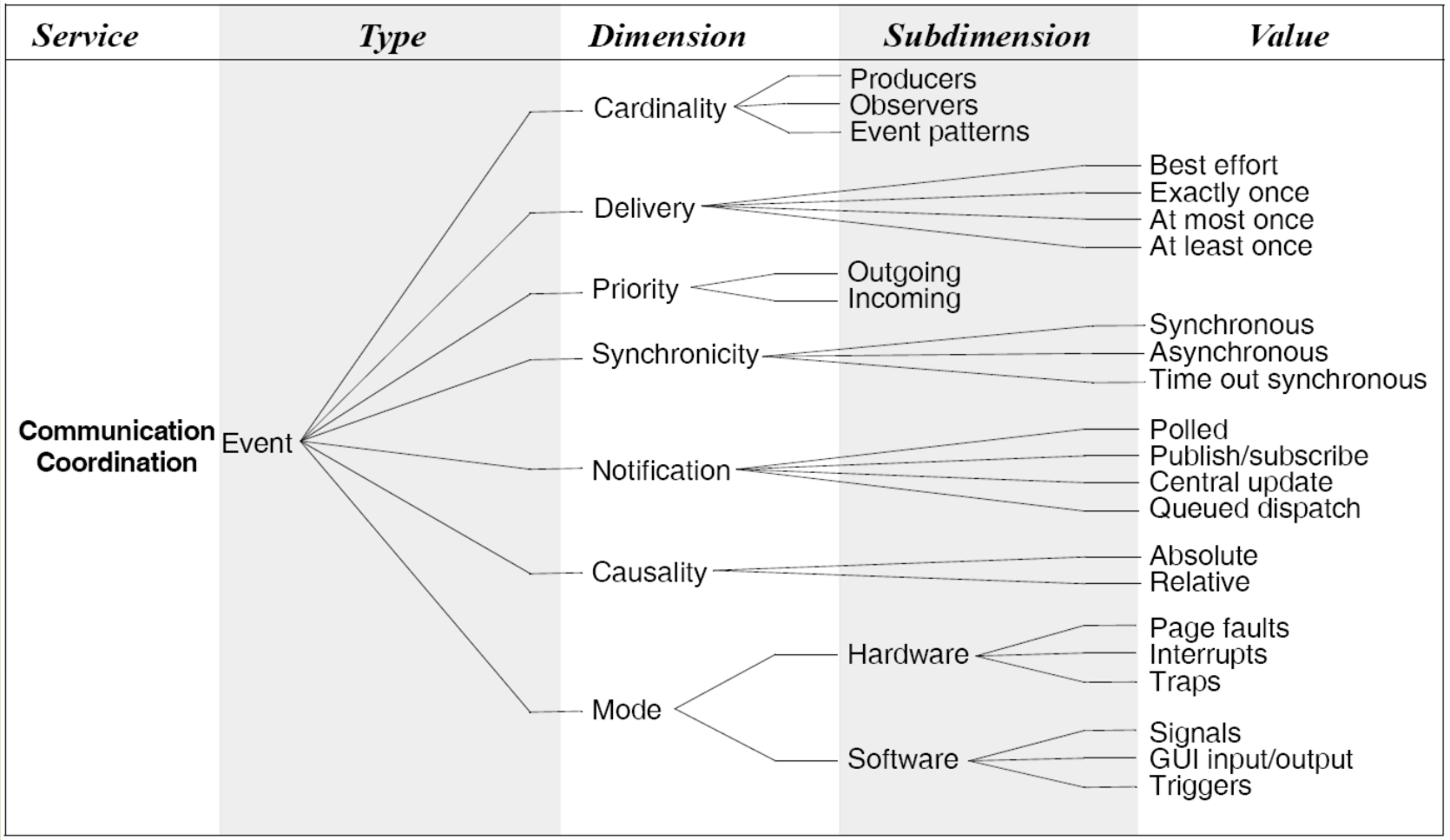
Event Connectors

- They are similar to procedure call connectors, in that they affect the flow of control among components, thus providing coordination services
- Once an event connector learns about the occurrence of an event, it generates event notifications to all interested parties and yields control to the components for processing these events
- Components can react at the occurrence of a single event or a combination of them
- Event connectors also provide communication services, in the sense that an event may carry information (e.g., time and place of occurrence)

Event Connectors (cont'd)

- Event connectors are different from procedure calls in that virtual connectors can be formed
  - Components may dynamically register and unregister their interest to receive certain events
- In distributed systems, typical dimensions are causality, atomicity, synchronicity, etc.
- Also, events may be generated by hardware (e.g., interrupts, page faults, etc.) and processed by software
- The cardinality of a multicasting event connector will be a single producer and multiple observer components, delivery of data will be done once, and its synchronicity may be asynchronous

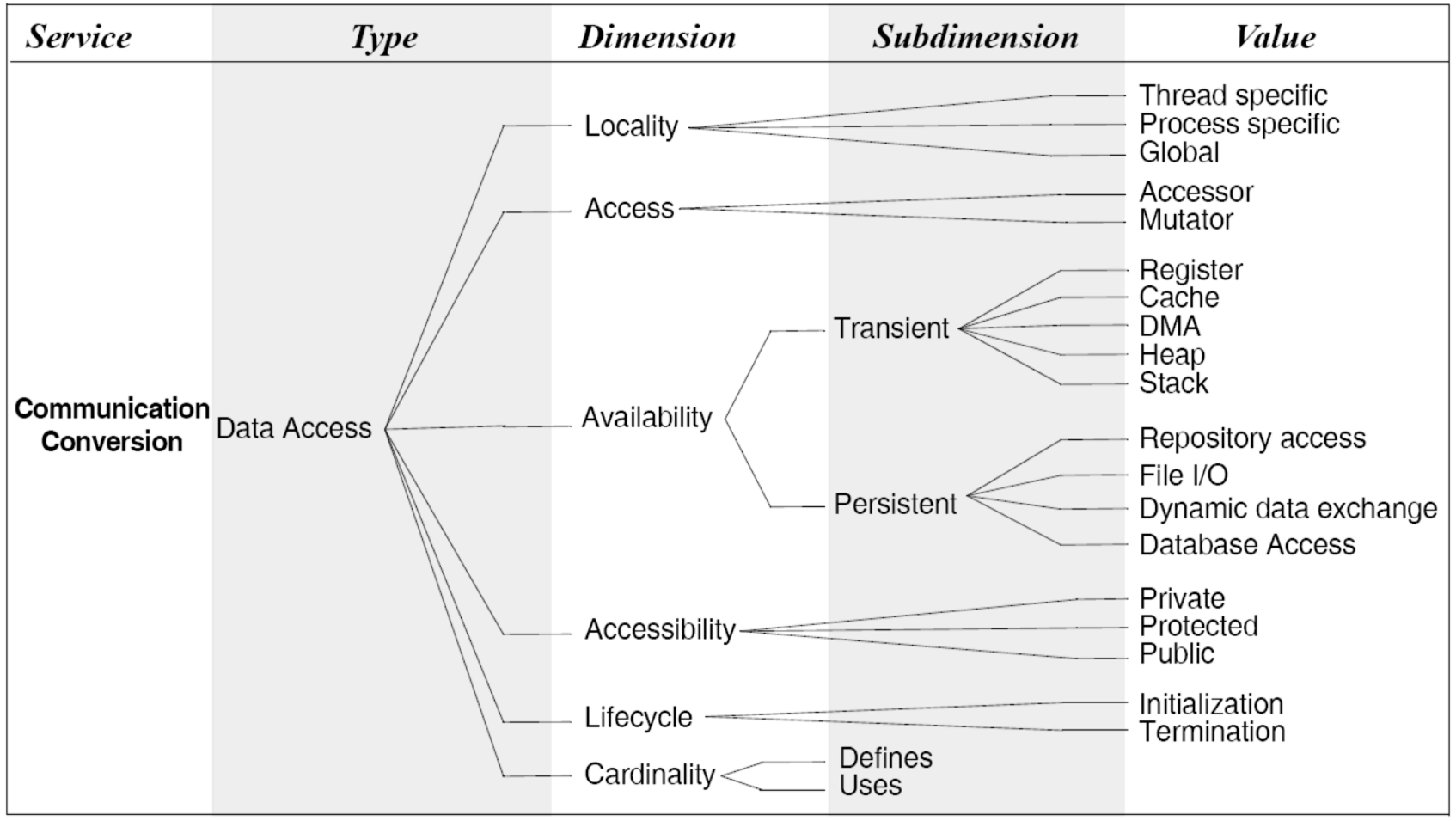
Event Connectors Type and Variations



Data Access Connectors

- They allow components to access data maintained by a data store component, so they provide communication services
- Data access may require preparation of the data store and translation in case of differences between the format of the required data and that of the stored data
- E.g., query mechanisms in SQL for database access, heap and stack memory access, and information caching
- Such a connector could enable global access, allow changing (mutating) of data, provide persistent access through file I/O, have a cardinality of 1 for defining the data and N for using the data, etc.

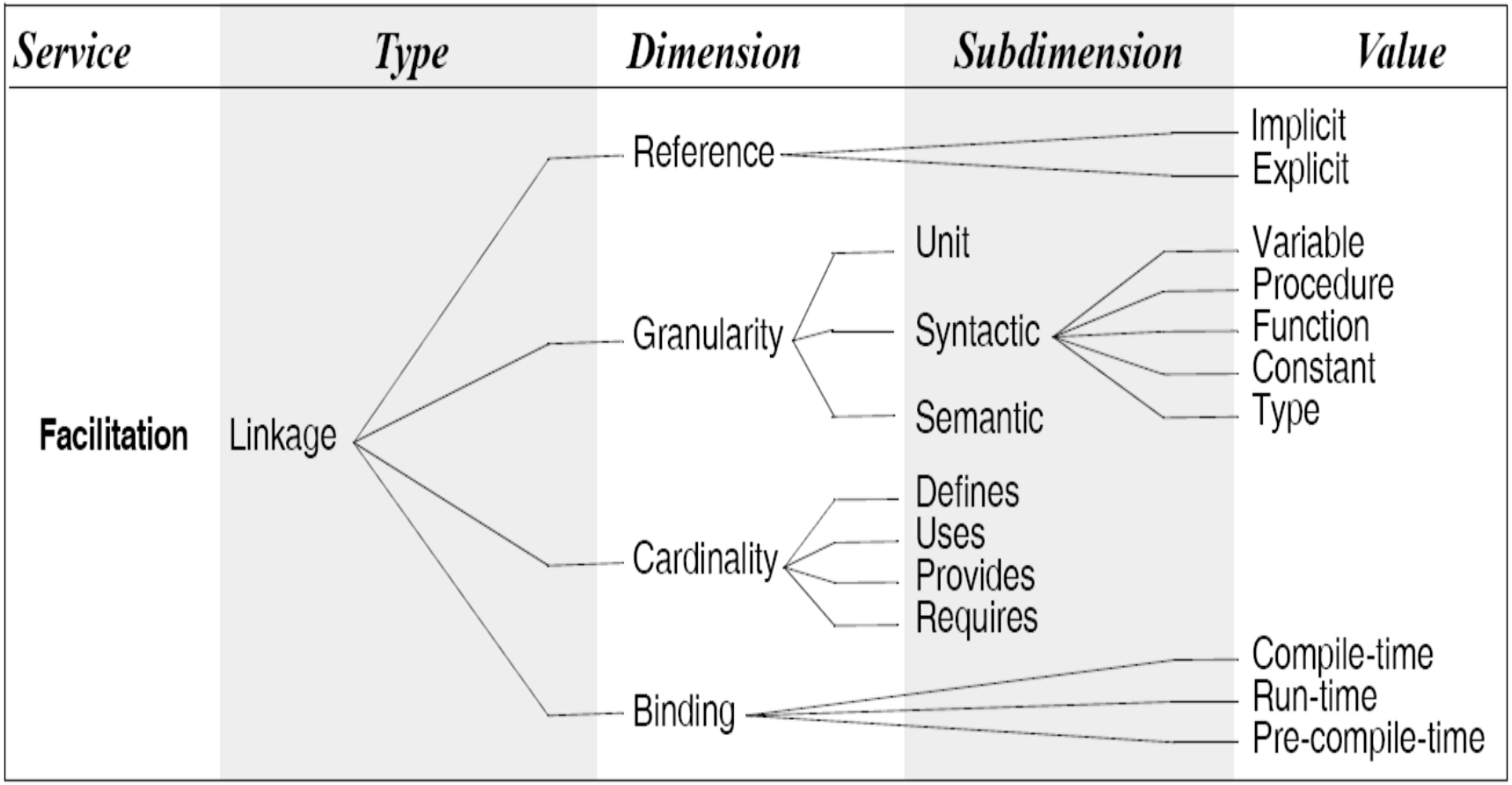
Data Access Connectors Type and Variations



Linkage Connectors

- They provide facilitation services, in the sense that they tie system components together and establish ducts
- Once ducts have been established, a linkage connector may disappear or remain in place to assist in the system's evolution
- Reference to linked components may be implicit or explicit
- Granularity refers to the size of components and level of detail to establish a linkage
  - *Unit* specifies only one component (e.g., `Make` in Unix)
  - *Syntactic* establishes links between variables, procedures, functions, constants, and types and can be used in static analysis
  - *Semantic* specifies how the linked components are supposed to interact

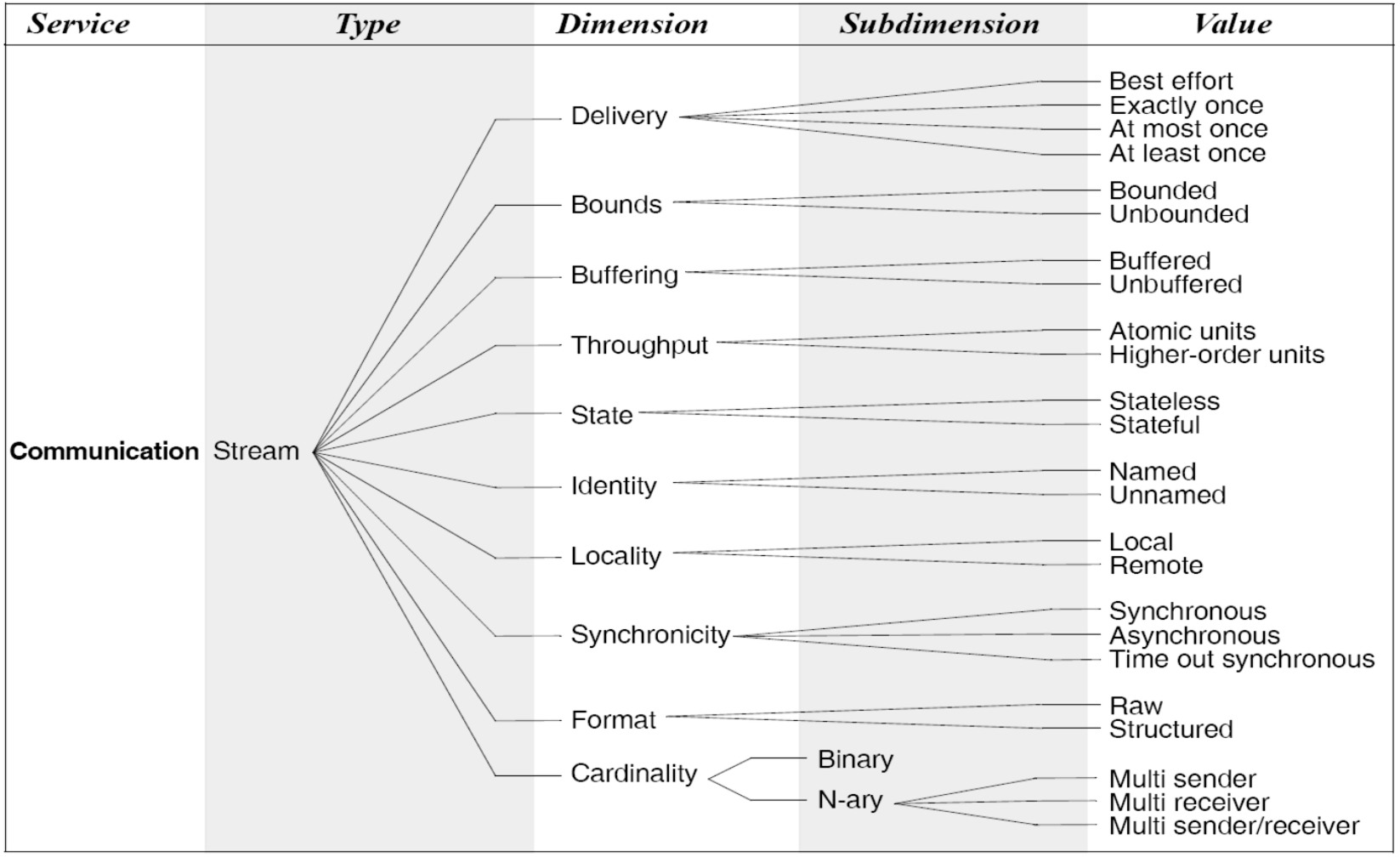
Linkage Connectors Type and Variations



Stream Connectors

- They provide communication services by means of transferring large amounts of data
- They can be combined with other connector types (e.g., data access connectors) to provide composite connectors for database and file storage access or event connectors to multiplex the delivery of a large number of events
- E.g., Unix pipes, TCP/IP sockets, client-server protocols
- A stream connector may be unnamed (as in Unix pipes), may provide asynchronous, remote interaction, may guarantee one delivery, may have cardinality 1 (single sender, single receiver, etc.)

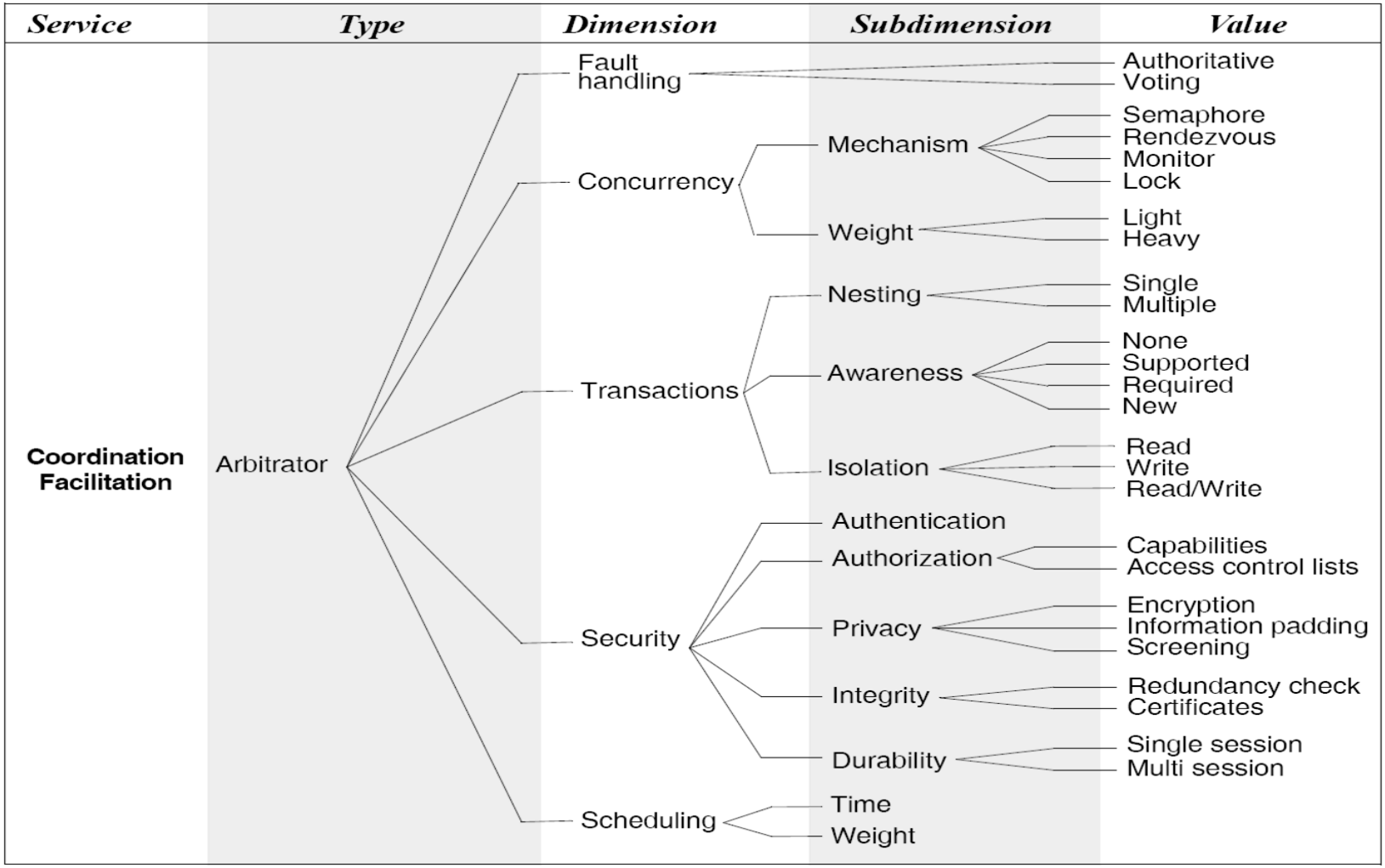
Stream Connectors Type and Variations



Arbitrator Connectors

- They provide facilitation services by streamlining system operations and resolving conflicts, typically in cases of component interaction where the components cannot make assumptions about the needs and state of other components
- They also provide coordination services, such as support for shared memory access supporting synchronization and concurrency control
- They also provide facilities for SLA negotiation, scheduling, and load balancing services, as well as support for reliability, safety and security

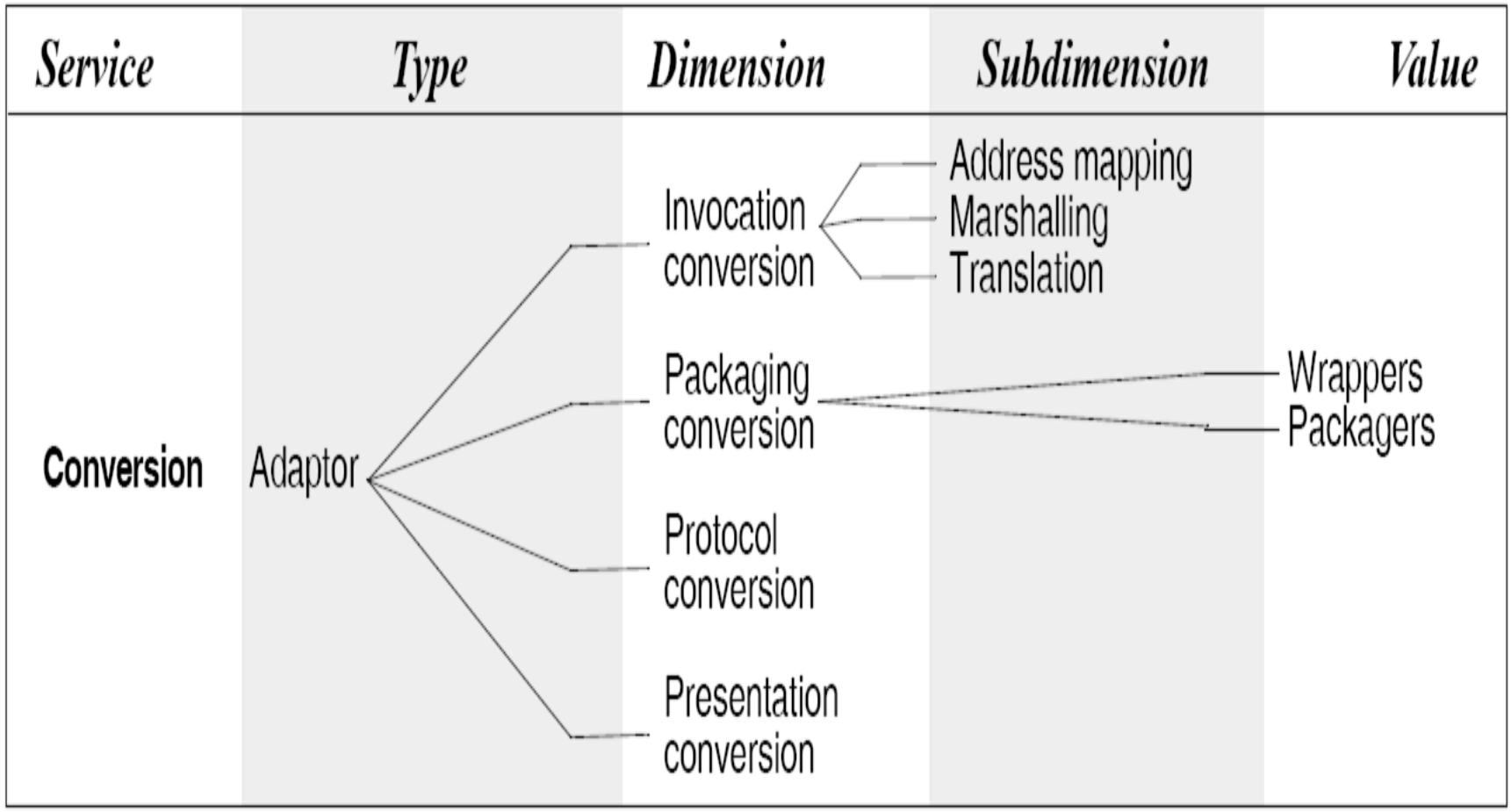
Arbitrator Connectors Type and Variations



Adaptor Connectors

- They provide facilities to support interaction between components that have not been designed to interoperate
- They provide conversion services in the form of matching communication policies and interaction protocols
- They are necessary for interoperation of components in heterogeneous environments (e.g., different programming languages or computer platforms)
- Optimization of component interaction can be performed by means of conversion
  - An RPC call converted to a local one, if the interacting components are in the same process

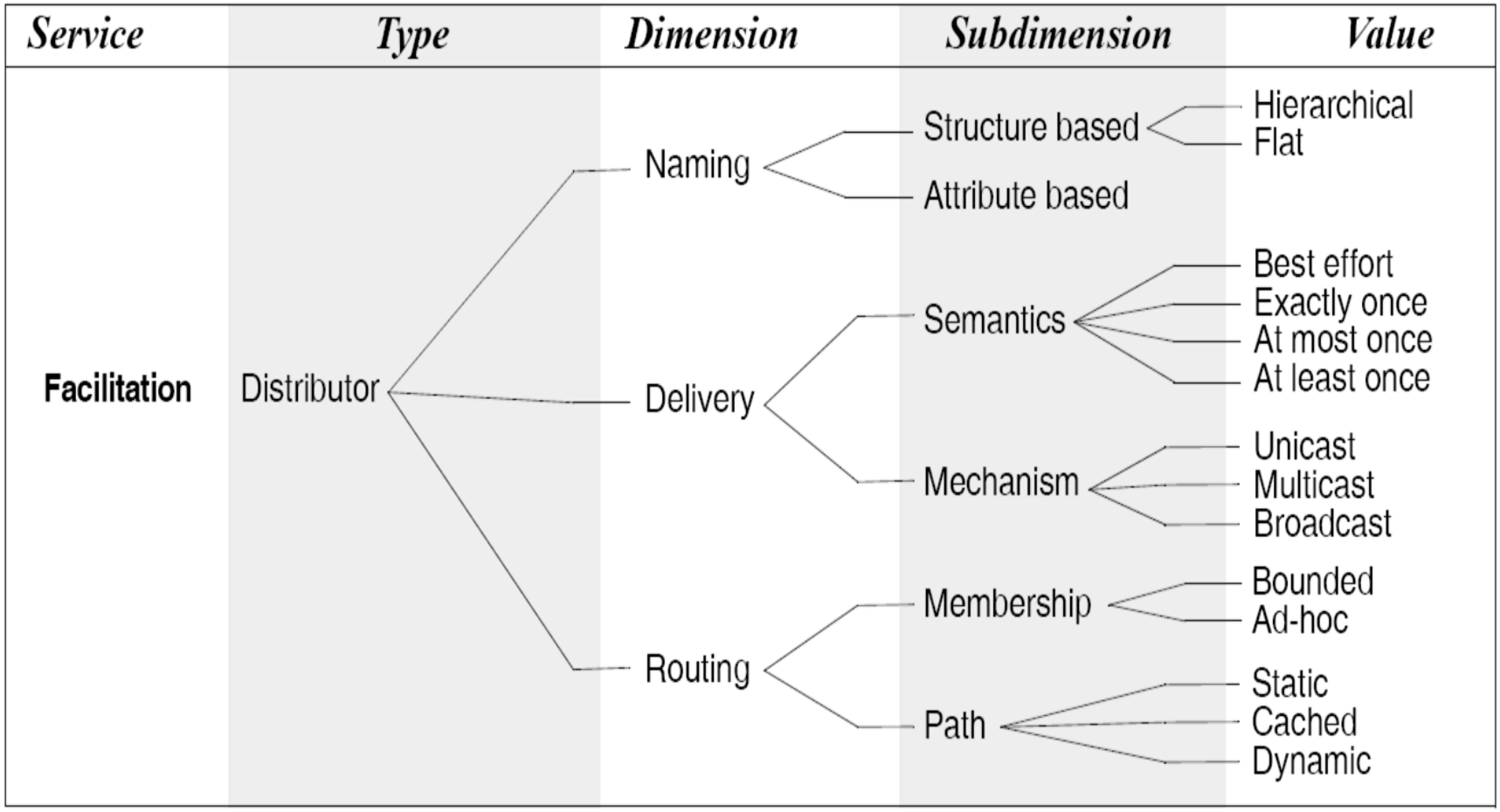
Adaptor Connectors Type and Variations



Distributor Connectors

- They provide facilitation services by performing the identification of interaction paths and subsequent routing of communication and coordination information among components
- They never exist by themselves, but provide assistance to other connectors, such as streams and procedure calls
- Distributed systems exchange information using distributor connectors
- E.g., DNS, routing, switching, etc.
- They play an important role in resource naming, scalability, survivability, data delivery, etc.

Distributor Connectors Type and Variations



Discussion

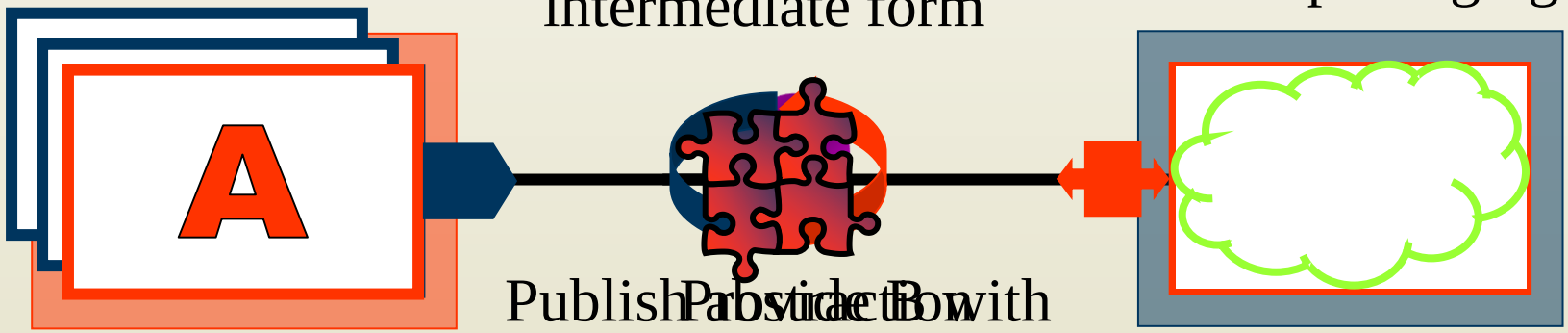
- Connectors allow modeling of arbitrarily complex interactions
- Connector flexibility aids system evolution
  - Component addition, removal, replacement, reconnection, migration
- Support for connector interchange is desired
  - Aids system evolution
  - May not affect system functionality

Discussion (cont'd)

- Libraries of OTS connector implementations allow developers to focus on application-specific issues
- Difficulties
  - Rigid connectors
  - Connector "dispersion" in implementations
- Key issue
  - Performance vs. flexibility

Role and Challenge of Software Connectors

How do we enable components A and B to interact?



How Does One Select a Connector?

- Determine a system's interconnection and interaction needs
  - Software interconnection models can help
- Determine roles to be fulfilled by the system's connectors
  - Communication, coordination, conversion, facilitation
- For each connector
  - Determine its appropriate type(s)
  - Determine its dimensions of interest
  - Select appropriate values for each dimension
- For multi-type, i.e., composite connectors
  - Determine the atomic connector compatibilities

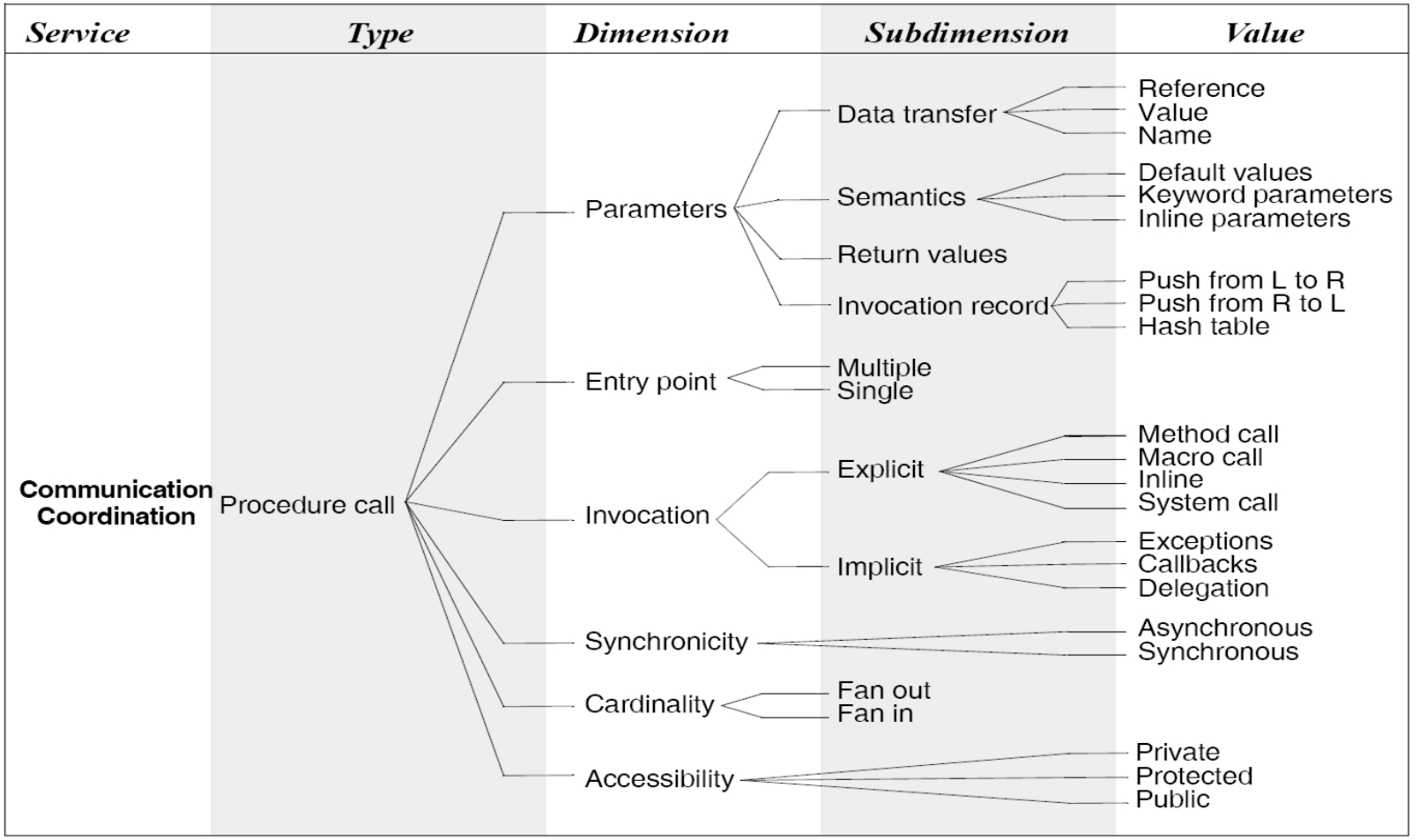
Simple Example

- System components will execute in two processes on the same host
  - Mostly intra-process
  - Occasionally inter-process
- The interaction among the components is synchronous
- The components are primarily computation-intensive
  - There are some data storage needs, but those are secondary

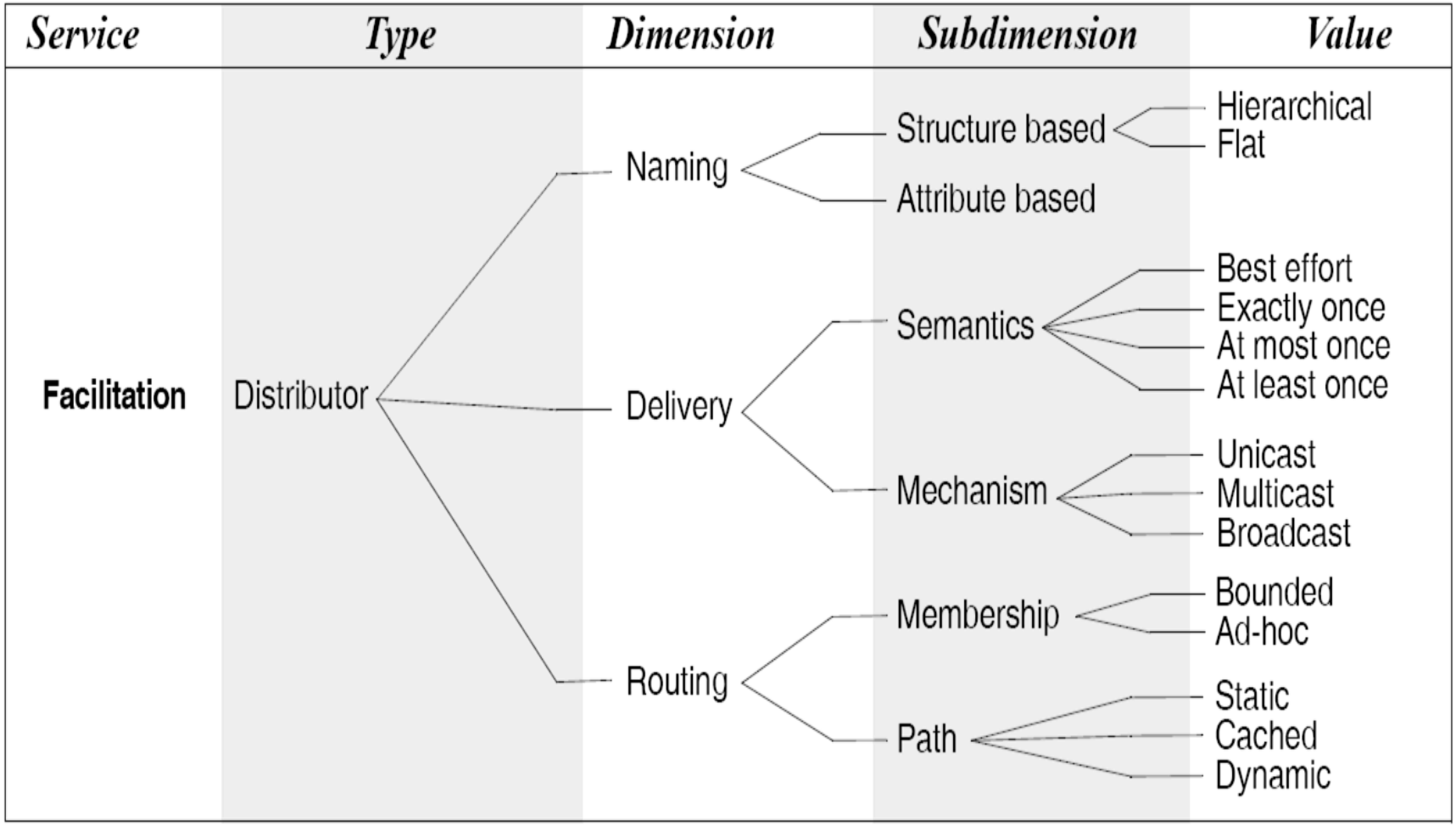
Simple Example (cont'd)

- Select procedure call connectors for intra-process interaction
- Combine procedure call connectors with distributor connectors for inter-process interaction
  - RPC
- Select the values for the different connector dimensions
  - What are the appropriate values?
  - What values are imposed by your favorite programming language(s)?

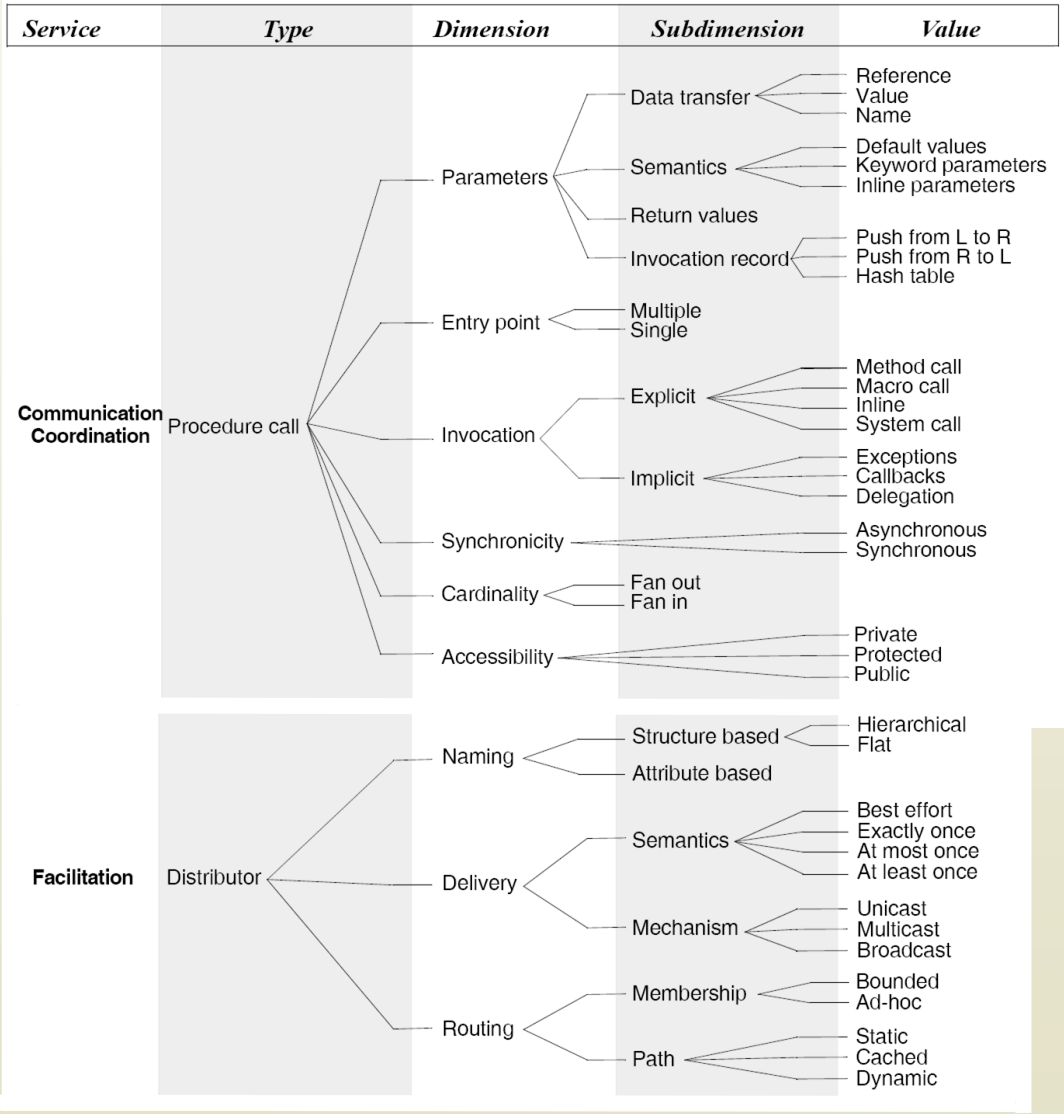
Procedure Call Connectors Revisited



Distributor Connectors Revisited



Two Connector Types in Tandem

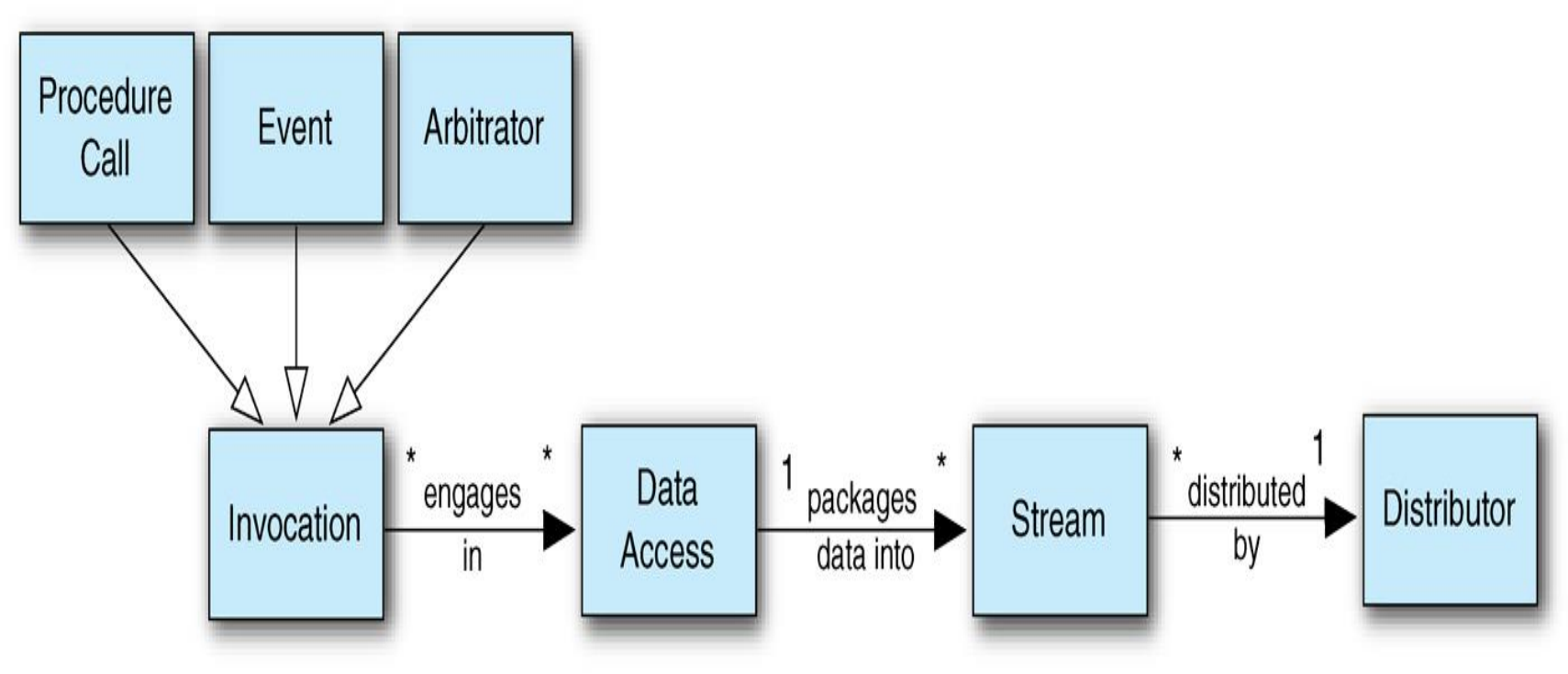


Select the appropriate values for PC and RPC!

Composite Distribution Connectors

- These connectors distribute large amounts of content (e.g., music, movies, scientific data, etc.) over, for example, Internet
- They can be described as different combinations of six of the connector types presented before
- Each distribution connector performs some form of data access, involving a stream-based reading/packaging of data and distribution of the latter to its users
- Some connector classes are invoked via procedure calls (e.g., client-server) while others are invoked via events or arbitration (e.g., P2P-based)

Composite Distribution Connectors Choices and Relationships



Software Interconnection Models Revisited

- Interconnection models (IM) as defined by Perry
  - Unit interconnection
  - Syntactic interconnection
  - Semantic interconnection
- All three are present in each system
- Are all equally appropriate at architectural level?

Unit Interconnection

- Defines relations between system's units
  - Units are components (modules or files)
  - Basic unit relationship is dependency
    - Unit-*IM* = ({units},{"depends on"})
- Examples
  - Determining context of compilation
    - e.g., C preprocessor
    - *IM* = ({files},{"include"})
  - Determining recompilation strategies
    - e.g., Make facility
    - *IM* = ({compile_units},{"depends on","has changed"})
  - System modeling
    - e.g., RCS, DVS, SVS, SCCS
    - *IM* = ({systems, files},{"is composed of"})

Unit Interconnection Characteristics

- Coarse-grain interconnections
  - At level of entire components
- Interconnections are static
- Does not describe component interactions
  - Focus is exclusively on dependencies

Syntactic Interconnection

- Describes relations among syntactic elements of programming languages
  - Variable definition/use
  - Method definition/invocation
    - *IM* = ({*methods*, *types*, *variables*, *locations*}, {"*is def at*", "*is set at*", "*is used at*", "*is del from*", "*is changed to*", "*is added to*"})
- Examples
  - Automated software change management
    - E.g., Interlisp's masterscope
  - Static analysis
    - E.g., Detection of unreachable code by compilers
  - Smart recompilation
    - Changes inside unit ➔ recompilation of only the changes
  - System modeling
    - Finer level of granularity than unit-IM

Syntactic Interconnection Characteristics

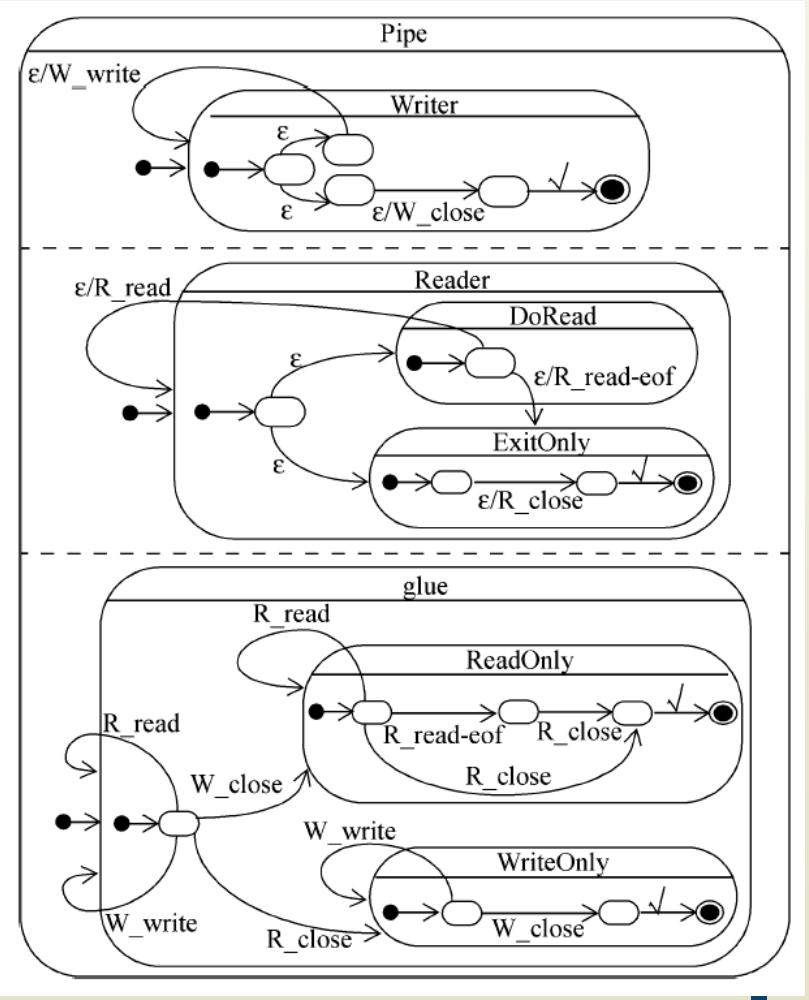
- Finer-grain interconnections
  - At level of individual syntactic objects
- Interconnections are static & dynamic
- Incomplete interconnection specification
  - Valid syntactic interconnections may not be allowed by semantics
  - Operation ordering, communication transactions
    - E.g., Pop on an empty stack
  - Violation of (intended) operation semantics
    - E.g., Trying to use calendar ***add*** operation to add integers

Semantic Interconnection

- Expresses how system components are meant to be used
  - Component designers' intentions
- Captures how system components are actually used
  - Component users' (i.e., system builders') intention
- Interconnection semantics can be formally specified
  - Pre- & post-conditions
  - Dynamic interaction protocols (e.g., CSP, FSM)
    - *IM* = ({*methods*, *types*, *variables*, ..., *predicates*}, {"*is set at*", "*is used at*", "*calls*", "*called by*", ..., "*satisfies*"})

Example of Semantic Interconnection

```
connector Pipe =
  role Writer = write → Writer □ close → ✓
  role Reader =
    let ExitOnly = close → ✓
    in let DoRead = (read → Reader
                    □ read-eof → ExitOnly)
    in DoRead □ ExitOnly
  glue = let ReadOnly = Reader.read → ReadOnly
                    □ Reader.read-eof
                                → Reader.close → ✓
                    □ Reader.close → ✓
     in let WriteOnly = Writer.write → WriteOnly
                     □ Writer.close → ✓
    in Writer.write → glue
     □ Reader.read → glue
     □ Writer.close → ReadOnly
     □ Reader.close → WriteOnly
```



Semantic Interconnection Characteristics

- Builds on syntactic interconnections
- Interconnections are static & dynamic
- Complete interconnection specification
  - Specifies both syntactic & semantic interconnection validity
- Necessary at level of architectures
  - Large components
  - Complex interactions
  - Heterogeneity
  - Component reuse
- What about ensuring other properties of interaction?
  - Robustness, reliability, security, availability, ...

Composing Basic Connectors

- In many systems a connector of multiple types may be required to service (a subset of) the components
- All connectors cannot be composed
  - Some are naturally interoperable
  - Some are incompatible
  - All are likely to require trade-offs
- The composition can be considered at the level of connector type dimensions and subdimensions

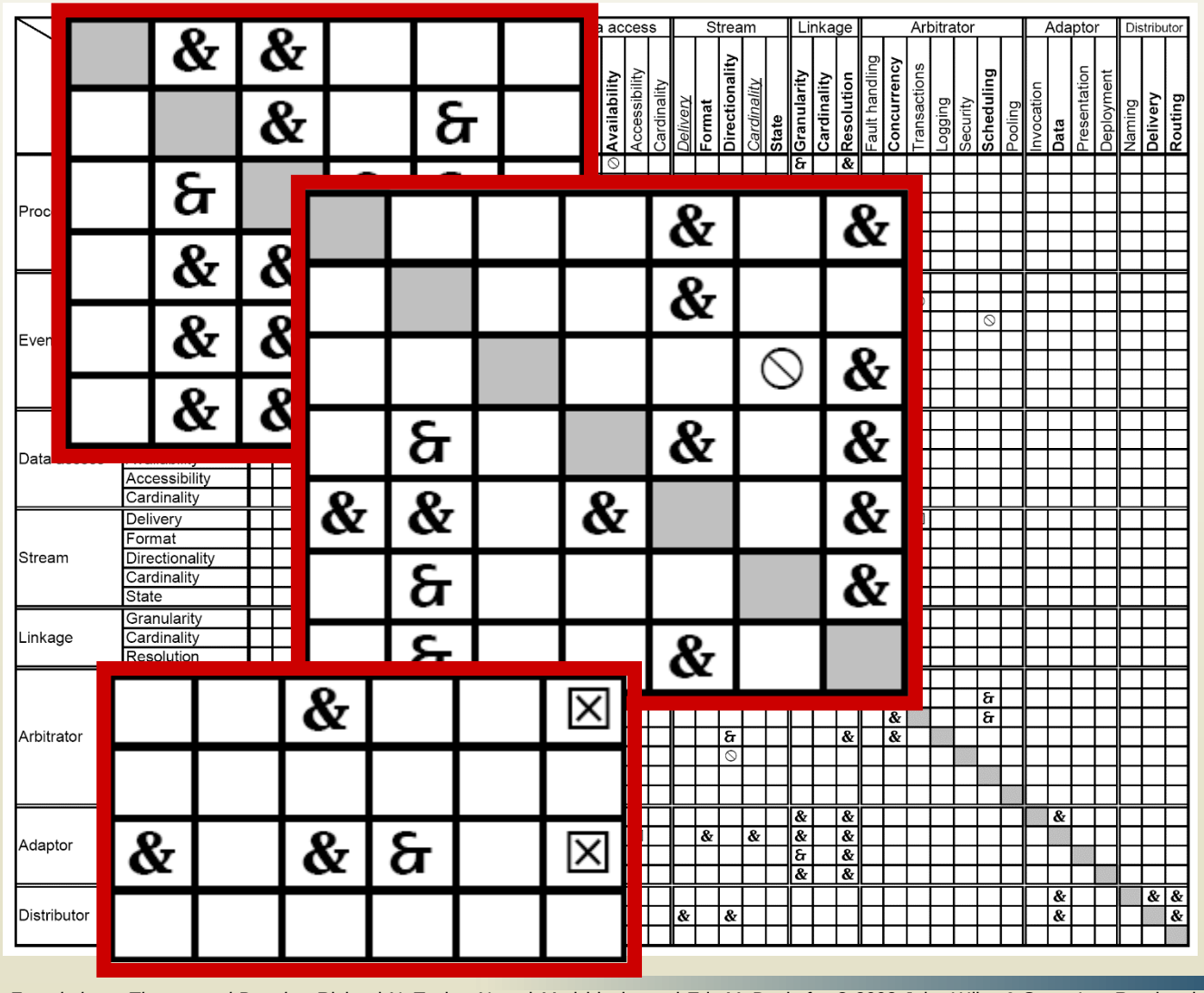
Selecting Appropriate Connectors

1. Select the specific set of interacting components; focus only on those components for which the desired connector is needed
2. Determine the interaction services the components need by studying the components' architectural descriptions and considering implementation aspects
3. Determine a subset of the eight connector types that comprise the initial candidate set for providing these services
4. Evaluate each connector type from the chosen subset based on the details of the interaction requirements and eliminate those that result in a suboptimal interaction solution
5. For each of the remaining candidate connector types, set the values for the necessary dimensions and subdimensions and identify the best (most natural) candidate connectors

Connector Dimension Inter-Relationships

- Requires – 
  - Choice of one dimension mandates the choice of another
- Prohibits – 
  - Two dimensions can never be composed into a single connector
- Restricts – 
  - Dimensions are not always required to be used together
  - Certain dimension combinations may be invalid
- Cautions – 
  - Combinations may result in unstable or unreliable connectors

Dimension Inter-Relationships



Well Known Composite Connectors

- Grid connectors (e.g., Globus)
  - Procedure call
  - Data access
  - Stream
  - Distributor
- Peer-to-peer connectors (e.g., Bittorrent)
  - Arbitrator
  - Data access
  - Stream
  - Distributor
- Client-server connectors
- Event-based connectors

Summary

- Every software system employs connectors, frequently many of them
- In complex (distributed) systems, the connectors may determine whether the desired system properties will be met
- At the most basic level, each connector establishes a conduit/channel/duct between two or more components to interact by exchanging data and/or control
- The space of connectors can be better understood by considering the role(s) played by a given connector and the type(s) of interaction supported by the connector
- At the same time, the large number of variation points for each connector type (dimensions, subdimensions, values) require that the (in)compatibilities among connectors be considered carefully