Systems for Design and Implementation

2015-2016

Course 9

Contents

CORBA

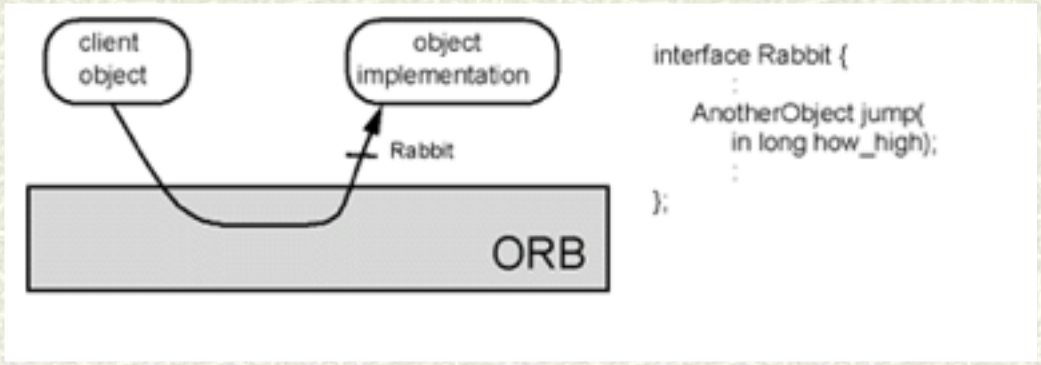
- CORBA (Common Object Request Broker Architecture) is a standard architecture for distributed object systems. It allows a distributed, *heterogeneous* collection of objects to interoperate.
- It was specified by the OMG (Object Management Group).
- CORBA products provide a framework for the development and execution of distributed applications.

Architecture

- CORBA defines an architecture for distributed objects.
- The basic CORBA paradigm is that of a request for services of a distributed object.
- Everything else defined by the OMG is in terms of this basic paradigm.
- The services that an object provides are given by its interface. Interfaces are defined in OMG's Interface Definition Language (IDL).
- Distributed objects are identified by object references, which are typed by IDL interfaces.

Architecture

- A client holds an object reference to a distributed object.
- The object reference is typed by an interface.
- The Object Request Broker, or ORB, delivers the request to the object and returns any results to the client.



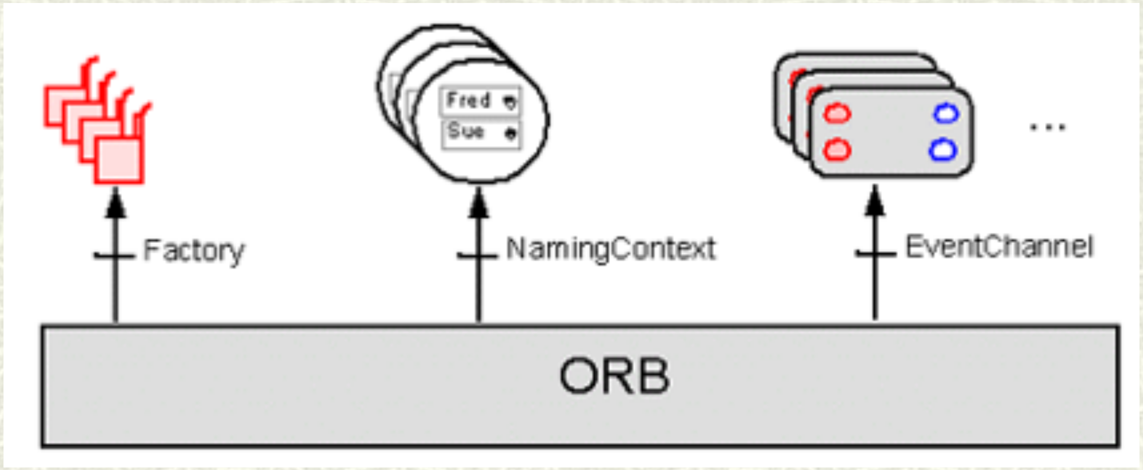
```
interface Rabbit {
    :
    AnotherObject jump(
        in long how_high);
    :
};
```

ORB

- The ORB is the distributed service that implements the request to the remote object. It locates the remote object on the network, communicates the request to the object, waits for the results and, when available, communicates those results back to the client.
- The ORB implements location transparency. Exactly the same request mechanism is used by the client and the CORBA object regardless of where the object is located.
- The ORB implements programming language independence for the request. The client issuing the request can be written in a different programming language from the implementation of the CORBA object. The ORB does the necessary translation between programming languages. Language bindings are defined for all popular programming languages.
- CORBA defines a network protocol, called IIOP (Internet Inter-ORB Protocol), that allows clients using a CORBA object to communicate with other CORBA objects. IIOP works across any TCP/IP implementation.

CORBA Services

- CORBA contains the definition of a set of distributed services to support the integration and interoperation of distributed objects, known as CORBA Services or COS.
- They are defined on top of the ORB as standard CORBA objects with IDL interfaces.



CORBA Services

Object life cycle	Defines how CORBA objects are created, removed, moved, and copied.
Naming	Defines how CORBA objects can have friendly symbolic names.
Events	Decouples the communication between distributed objects.
Transactions	Coordinates atomic access to CORBA objects.
Concurrency Control	Provides a locking service for CORBA objects in order to ensure serializable access.
Query	Supports queries on objects.

Object Adapters

- An Object Adapter is a server-side facility that provides a mechanism for the CORBA object implementations to communicate with the ORB and vice versa.
- An Object Adapter also extends the functionality of the ORB. An Object Adapter is layered on top of the ORB to provide an interface between the ORB and the object implementation.
- Object Adapters can also be used to provide specialized services optimized for a particular environment, platform, or object implementation.
- Some of the services provided by an Object Adapter are:
  - Registration of server object implementations.
  - Activation and deactivation of object implementations.
  - Instantiation of objects at runtime.
  - Generation and management of object references.
  - Mapping of object references to their implementations.
- While many object adapter implementations may exist for unique situations, the CORBA specification only requires implementations to provide a Basic Object Adapter (BOA) or a Portable Object Adapter (POA) which is a portable version of the BOA.

IDL Interfaces

- The OMG Interface Definition Language IDL supports the specification of object interfaces.
- An object interface indicates the operations the object supports, but not how they are implemented.
- In IDL you cannot declare object state and algorithms.
- The implementation of a CORBA object is provided in a standard programming language, such as the Java programming language or C++. An interface specifies the contract between the code using the object and the code implementing the object. Clients only depend on the interface.
- IDL interfaces are programming language neutral. IDL defines language bindings for many different programming languages. This allows an object implementor to choose the appropriate programming language for the object. Similarly, it allows the developer of the client to choose the appropriate and possibly different programming language for the client.
- Language bindings for the C, C++, Java, Ada, Smalltalk, and other programming languages.

IDL

- The IDL data types are:
  - Basic data types (long, short, string, float...)
  - Constructed data types (struct, union, enum, sequence)
  - Typed object references
  - The *any* type, a dynamically typed value
- Using IDL the following can be described:
  - Modularized object interfaces
  - Operations and attributes that an object supports
  - Exceptions raised by an operation
  - Data types of an operation return value, its parameters, and an object's attributes
- The operations parameters are tagged with the keywords *in*, *out*, or *inout*.
  - The *in* keyword indicates that the data is passed from the client to the object.
  - The *out* keyword indicates that the data is returned from the object to the client,
  - The *inout* keyword indicates that the data is passed from the client to the object and then returned to the client.

IDL Example

```
module BookShop {
  struct Book {
      double price;
      String title;
      String author
  };
  exception Unknown{};
  interface Library {
      Book getBook(in string title) raises(Unknown);
      readonly attribute string name;


  };
  interface BookFactory {
    Book create_book( in string title, in string author, in double
   price);
  };
};
```

IDL to Java Binding

IDL	Java
module	package
interface	interface
operation	method
attribute	pair of methods (get/set)
exception	exception
struct/union	final class

IDL to Java Binding

IDL Type	Java Type
boolean	boolean
char / wchar	char
octet	byte
short / unsigned short	short
long / unsigned long	int
long long / unsigned long long	long
float	float
double	double
string / wstring	String

Mapping Parameter Passing Modes

- IDL *in* parameters are mapped to normal Java actual parameters. The results of IDL operations are returned as the result of the corresponding Java method.

- IDL *out* and *inout* parameters cannot be mapped directly into the Java parameter-passing mechanism. This mapping defines additional holder classes for all the IDL basic and user-defined types used to implement these parameter modes in Java.

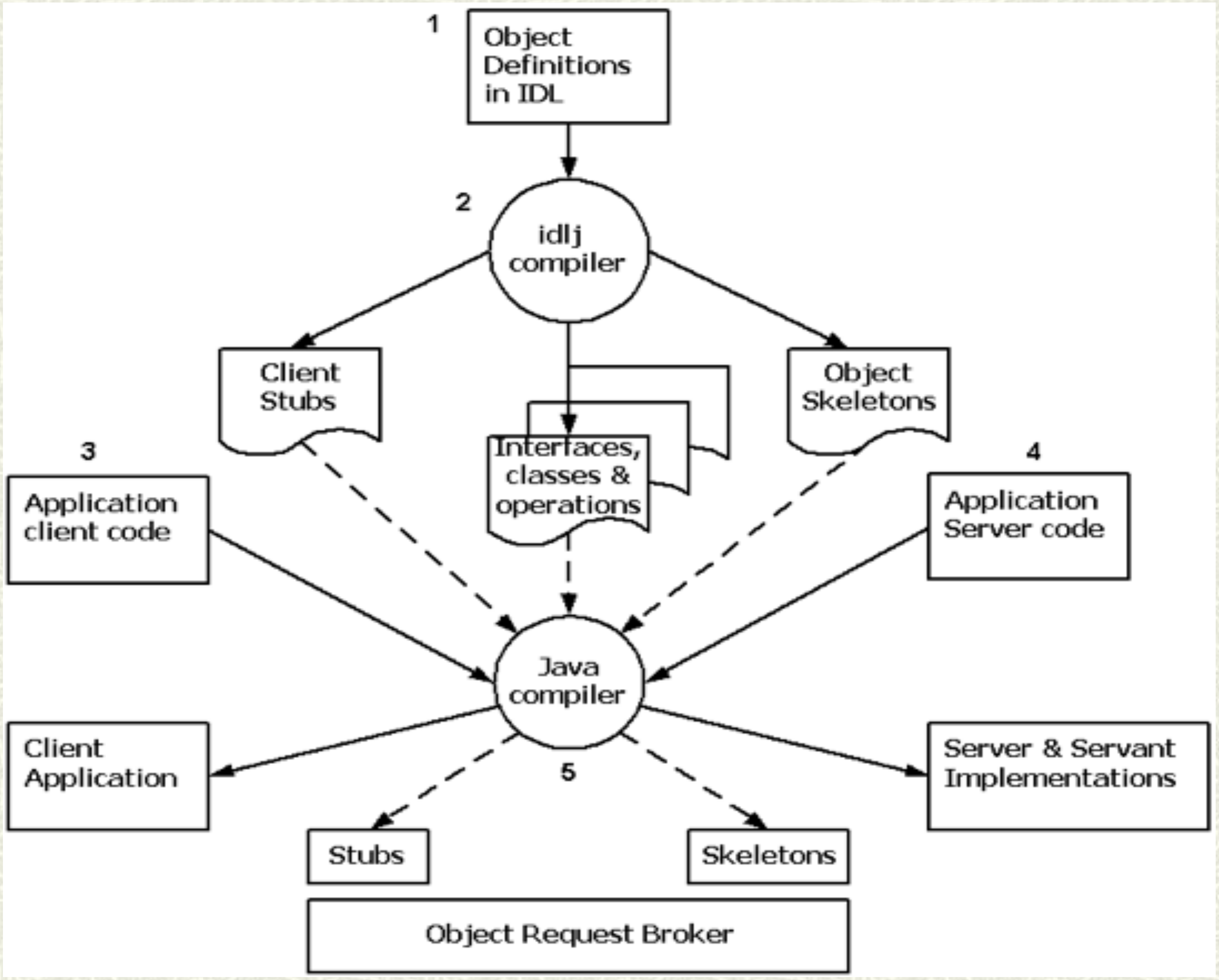
Example: boolean

- in: **`boolean`**
- out/inout: **`BooleanHolder`**
- return value: **`Boolean`**

Mapping Exceptions

- A user-defined exception is mapped to a final Java class that extends `org.omg.CORBA.UserException`.
- The `UserException` class extends the standard `org.omg.CORBAexception` class. The mapping is identical to the IDL struct type, including generated `Holder` and `Helper` classes.
- The standard IDL system exceptions are mapped to final Java classes that extend `org.omg.CORBA.SystemException` and provide access to the IDL major and minor exception code, as well as a string describing the reason for the exception.
- Instantiating `org.omg.CORBA.SystemException` is not possible. Only classes that extend it can be instantiated.

CORBA and Java



Steps for development

1. Write a specification for each object using the IDL.
2. Use the IDL compiler to generate the client stub code and server skeleton code.
3. Write the client application code.
4. Write the server object code.
5. Compile the client and server code.
6. Start the server.
7. Run the client application.

Java API

- **`org.omg.CORBA`** provides the mapping of the OMG CORBA APIs to the Java programming language, including the class ORB, which is implemented so that a programmer can use it as a fully-functional Object Request Broker (ORB).
- **`org.omg.CosNaming`** provides a naming service for Java IDL

Thrift

- Thrift is a software library and set of code-generation tools developed at Facebook to expedite development and implementation of efficient and scalable backend services.
- It allows developers to define datatypes and service interfaces in a single language-neutral file and generate all the necessary code to build RPC clients and servers.
- It combines a language-neutral software stack implemented across numerous programming languages and an associated code generation engine that transforms a simple interface and data definition language into client and server remote procedure call libraries.
- Key components:
  - types
  - transport
  - protocol
  - versioning
  - processors

Thrift Types

- The Thrift type system does not introduce any special dynamic types or wrapper objects. It also does not require that the developer write any code for object serialization or transport.
- Base types, structs, containers.
- The type system rests upon a few base types:
  - *bool* - boolean value, true or false
  - *byte* - a signed byte
  - *i16* - 16-bit signed integer
  - *i32* - 32-bit signed integer
  - *i64* - 64-bit signed integer
  - *double* - 64-bit floating point number
  - *string*
  - *binary* - byte array representation for blobs

Thrift Types -Structs

- A Thrift struct defines a common object to be used across languages.
- A struct is essentially equivalent to a class in object oriented programming languages.
- A struct has a set of strongly typed fields, each with a unique name identifier.
- The basic syntax for a struct looks very similar to a C struct definition.
- Fields may be annotated with an integer field identifier (unique to the scope of that struct) and optional default values.
- Field identifiers will be automatically assigned if omitted.

```
struct Example {
  1:i32 number=10,
  2:i64 bigNumber,
  3:double decimals,
  4:string name="thrifty"
}
```

Thrift Types -Containers

- Thrift containers are strongly typed containers that map to the most commonly used containers in common programming languages.
- They are annotated using the C++ template (or Java Generics) style.
- There are three types available:
  - `list<type>` An ordered list of elements.
    - Translates directly into an STL `vector`, Java `ArrayList`, or native array in scripting languages. May contain duplicates.
  - `set<type>` An unordered set of unique elements.
    - Translates into an STL `set`, Java `HashSet`, `set` in Python, or native dictionary in PHP/Ruby.
  - `map<type1,type2>` A map of strictly unique keys to values
    - Translates into an STL `map`, Java `HashMap`, PHP associative array, or Python/ Ruby dictionary.
- Container elements may be of any valid Thrift type, including other containers or structs.
- In the target language, each definition generates a type with two methods, `read` and `write`, which perform serialization and transport of the objects using a Thrift TProtocol object.

Thrift - Exceptions

- Exceptions are syntactically and functionally equivalent to structs except that they are declared using the `exception` keyword instead of the `struct` keyword.
- The generated objects inherit from an exception base class as appropriate in each target programming language, in order to seamlessly integrate with native exception handling in any given language.

Thrift - Services

- Services are defined using Thrift types.
- Definition of a service is semantically equivalent to defining an interface (or a pure virtual abstract class) in object oriented programming.
- The Thrift compiler generates fully functional client and server stubs that implement the interface.
- Argument lists and exception lists for functions are implemented as Thrift structs. All three constructs are identical in both notation and behavior.

```
service <name> {
  <returntype> <name>(<arguments>)
    [throws (<exceptions>)]
  ...
}


service StringCache {
   void set(1:i32 key, 2:string value),
   string get(1:i32 key) throws (1:KeyNotFound knf),
   void delete(1:i32 key)
}
```

Thrift - Transport

- The transport layer is used by the generated code to facilitate data transfer.
- Thrift is typically used on top of the TCP/IP stack with streaming sockets as the base layer of communication.
- Generated Thrift code only needs to know how to read and write data.
- The origin and destination of the data are irrelevant; it may be a socket, a segment of shared memory, or a file on the local disk.
- The Thrift `TTransport` interface supports the following methods:
    - `open` Opens the transport
    - `close` Closes the transport
    - `isOpen` Indicates whether the transport is open
    - `read` Reads from the transport
    - `write` Writes to the transport
    - `flush` Forces any pending writes

Thrift - Transport

- There is also a `TServerTransport` interface used to accept or create primitive transport objects.
  - `open` Opens the transport
  - `listen` Begins listening for connections
  - `accept` Returns a new client transport
  - `close` Closes the transport
- The transport interface is designed for simple implementation in any programming language.
- New transport mechanisms can be easily defined as needed.
  - The `TSocket` class is implemented across all target languages. It provides a common, simple interface to a TCP/IP stream socket.
  - The `TFileTransport` is an abstraction of an on-disk file to a data stream. It can be used to write out a set of incoming Thrift requests to a file on disk.

Thrift - Protocol

- Thrift enforces a certain messaging structure when transporting data, but it is oblivous to the protocol encoding in use.
- It does not matter whether data is encoded as XML, human-readable ASCII, or a dense binary format as long as the data supports a fixed set of operations that allow it to be deterministically read and written by generated code.
- The Thrift Protocol interface supports:
  - bidirectional sequenced messaging,
  - encoding of base types, containers, and structs.
- It has implemented and deployed a space-efficient binary protocol.
- It writes all data in a flat binary format:
  - integer types are converted to network byte order,
  - strings are prepended with their byte length,
  - message and field headers are written using the primitive integer serialization constructs.
  - String names for fields are omitted - when using generated code, field identifiers are sufficient.

Thrift - Protocol

```
writeMessageBegin(name, type, seq)
writeMessageEnd()
writeStructBegin(name)
writeStructEnd()
writeFieldBegin(name, type, id)
writeFieldEnd()
writeMapBegin(ktype, vtype, size)
writeMapEnd()
...
name, type, seq = readMessageBegin();
readMessageEnd()
name = readStructBegin()
readStructEnd()
name, type, id = readFieldBegin()
 readFieldEnd()
k, v, size = readMapBegin()
readMapEnd()
...
```

Thrift - Versioning

- Thrift is robust in the face of versioning and data definition changes.
- Versioning is implemented via field identifiers.
- The field header for every member of a struct in Thrift is encoded with a unique field identifier.
- The combination of this field identifier and its type specifier is used to uniquely identify the field.
- The Thrift definition language supports automatic assignment of field identifiers, but it is recommended to always explicitly specify field identifiers.
- When an unexpected field is encountered, it can be safely ignored and discarded.
- When an expected field is not found, a solution exists to signal to the developer that it was not present.
- This is implemented via an inner **`isset`** structure inside the defined objects.
- The inner isset object of each Thrift struct contains a boolean value for each field which denotes whether or not that field is present in the struct. When a reader receives a struct, it should check for a field being set before operating directly on it.

Thrift - RPC Implementation

- `TProcessor` instance capable of handling RPC requests to that service, using a few helpers

```
interface TProcessor {
bool process(TProtocol in, TProtocol out) throws TException
}
```

- The `TServer` object works as follows:
  - Use the `TServerTransport` to get a `TTransport.`
  - Use the `TTransportFactory` to optionally convert the primitive transport into a suitable application transport.
  - Use the `TProtocolFactory` to create an input and output protocol for the `TTransport`
  - `Invoke` the `process`() method of the `TProcessor` object
- The layers are appropriately separated such that the server code needs to know nothing about any of the transports, encodings, or applications in play.
- The server encapsulates the logic around connection handling, threading, etc. while the processor deals with RPC.
- The only code written by the application developer is in the Thrift file and the interface implementation.

SOA

- Service Oriented Architecture (SOA) is an alternative model for developing enterprise applications.
- SOA is an architectural style for building enterprise solutions based on services.
- It promotes the concept of a business-aligned enterprise service as the fundamental unit of designing, building, and composing enterprise business solutions.
- SOA describes the structure of services, their relationships, and the value they bring to enterprise processes and solutions.

The concept of service

- The fundamental concept in SOA is a service.
- A service is a discrete unit of business functionality that is made available through a service contract. The service contract specifies all interactions between the service consumer and service provider. This includes:
  - Service interface
  - Interface documents
  - Service policies
  - Quality of service (QoS)
  - Performance
- One of the main differences between a service and other software constructs (such as components or objects) is that a service is explicitly managed. The quality of service and performance are managed through a service level agreement. In addition, the entire service life cycle is managed from design, to deployment, to enhancements, to maintenance.

The concept of service

- A service specifically separates the interface from the implementation.
- The **service interface** specifies the service operations, that is, what the service does, the parameters that are passed into and out of the operation, and the protocols for how those capabilities are used and provided. A service typically contains several different, but related, operations.
- The **service implementation** is how the service provides the capabilities of its interface. The implementation may be based on existing applications, on using other services to combine their capabilities, on code written specifically for the service, or all of the above. *Consumers of the service* see only what the service does, not how it is implemented while the *producer of a service* is free to change the implementation of a service, as long as it does not change the interface or the behavior.

Characteristics of Services

- **Modularity and granularity** - Business processes are decomposed into modular services that are self-contained. Services themselves can be composed from other modular services, and can be mixed and matched as needed to create new composite services.
- **Granularity** refers to the size or amount of functionality in a given interaction and is a quality for functional improvement of a service:
  - *Coarse-grained* if they consist of fewer, larger components . If services are coarse-grained then they are capable of offering larger and richer functionality within a single service operation. This helps to reduce complexity and network overload by reducing the steps necessary to fulfill a given business activity.
  - *Fine-grained* service operations provide the exchange of small amounts of information to complete a specific discrete task.

Characteristics of Services

- **Encapsulation** -Services expose a strict separation of the service interface (what a service does) from the service implementation (how it is done). Encapsulation hides the service's internal implementation details and data structures from the published interface operations and semantic model.
- **Autonomy** - Autonomy is the characteristic that allows services to be deployed, modified, and maintained independently from each other and the solutions that use them. An autonomous service life cycle is independent of other services.
- **Reuse** - Modularity, encapsulation, autonomy enable services to be combined into multiple business processes or accessed by multiple service consumers from multiple locations and in multiple contexts. Services are shared and reused as building blocks in the construction of processes or composite services.

Characteristics of Services

- **Stateless** - Service operations are stateless. This means that they neither remember the last thing they were asked to do nor care what the next is. Services are not dependent on the context or state of other services - only on their functionality.
- **Self-describing** - The service contract provides a complete description of the service interface, its operations, the input and output parameters, and schema. The contract may also contain pre and postconditions and constraints about the operations.
- **Composable** - Services can be composed from other services and, in turn, can be combined with other services to compose new services or business processes.

Characteristics of Services

- **Governed by policy** - Relationships between service consumers and providers (and between services and service domains) are governed by policies and service level agreements (SLAs). Policies describe how different consumers are allowed to interact with the service -what they are allowed to do.
- **Independent of location, language, and protocol** - Services are designed to be location and protocol/platform-independent. They should be accessible to any authorized user, on any platform, from any location.

Programing Models For SOA

- **Service Component Architecture** (SCA).
  - It is a set of specifications that defines an assembly model for composite services and also provides a programming model to build applications which are SOA-based.
- **Service Data Objects** (SDO).
  - It aims to provide consistent means of handling data within applications, whatever the source or format may be.
  - Service Data Objects are designed to simplify and unify the way in which applications handle data.
  - Using SDO, application programmers can uniformly access and manipulate data from heterogeneous data sources, including relational databases, XML data sources, Web services, and enterprise information systems.

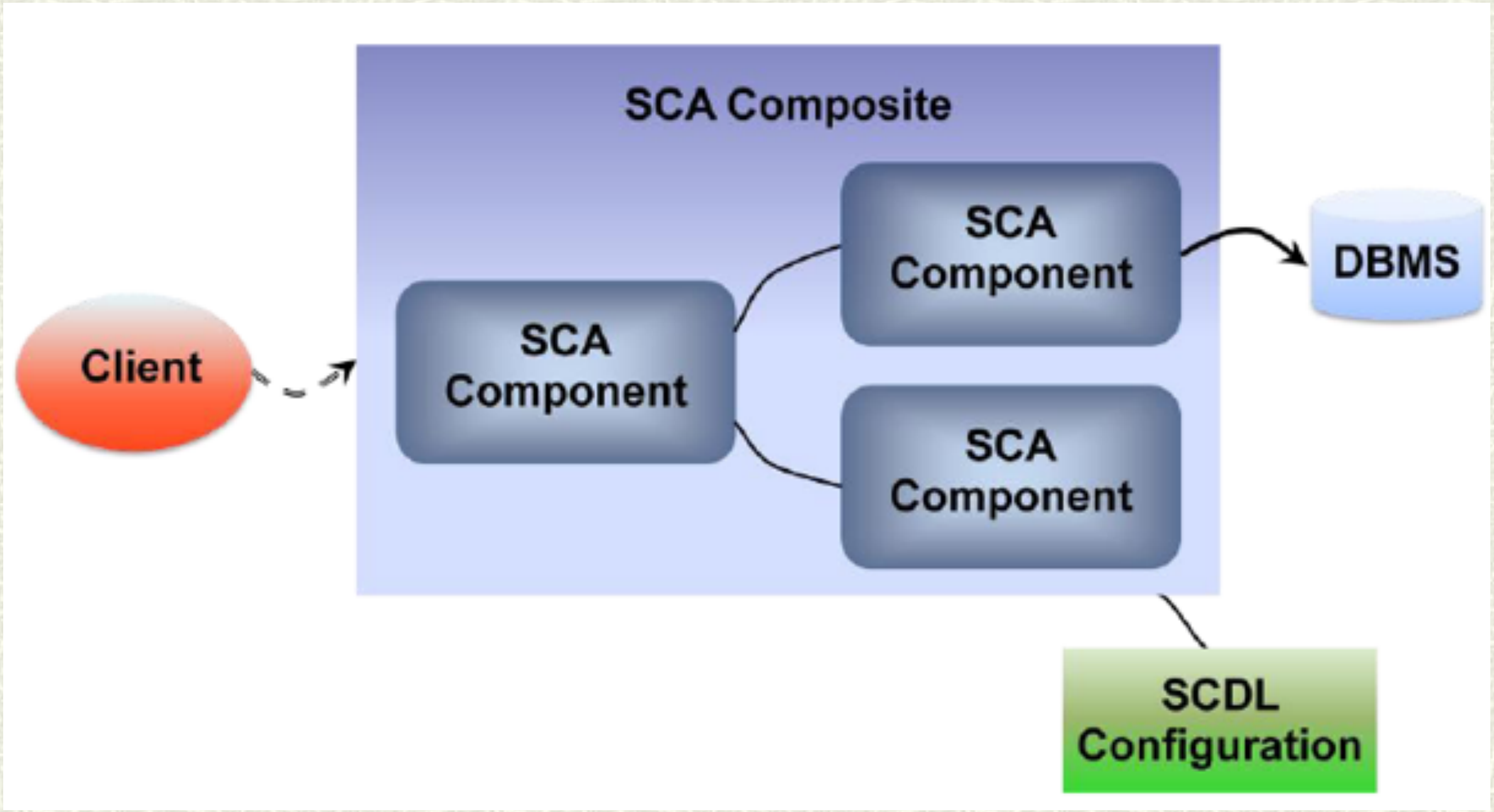
Service Component Architecture (SCA)

- The *SCA specifications* define how to create components and how to combine those components into complete applications.
- The components in an SCA application might be built with Java or other languages using SCA-defined programming models, or they might be built using other technologies, such as the Business Process Execution Language (BPEL) or the Spring Framework.
- Whatever component technology is used, SCA defines a common assembly mechanism to specify how those components are combined into applications.

SCA –Components and Composites

- Every SCA application is built from one or more components (classes in Java, C ++, BPEL).
- The components can be combined into larger structures called *composites*.
- A *composite* is a logical construct:
  - Its components can run in a single process on a single computer or be distributed across multiple processes on multiple computers.
  - A complete application might be constructed from just one composite, or it could combine several different composites.
  - The components making up each composite might all use the same technology, or they might be built using different technologies—either option is possible.

SCA –Components and Composites



SCA –Components and Composites

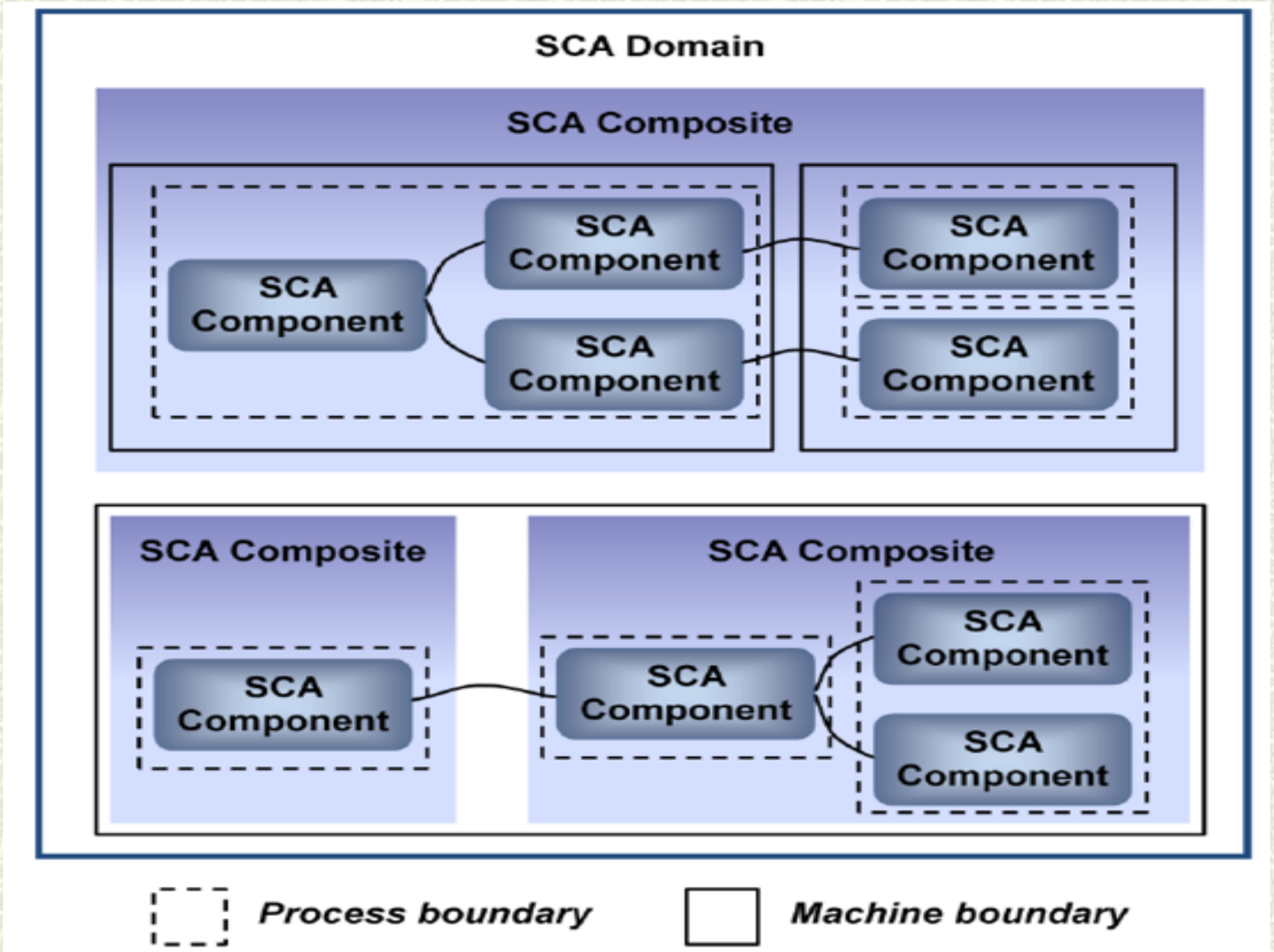
- An SCA application can be accessed by software from the non-SCA world, such as a JavaServer Page (JSP), a Web services client, or anything else.
- Components in an SCA application can also access data, just like any other application.
- An SCA composite is typically described in an associated configuration file, the name of which ends in *.composite*. This file uses an XML-based format called the Service Component Definition Language (SCDL) to describe the components this composite contains and specify how they relate to one another.

```
<composite name="ExampleComposite" ...>
      <component name="Component1"> ... </component>
      <component name="Component2"> ... </component>
      <component name="Component3"> ... </component>
 </composite>
```

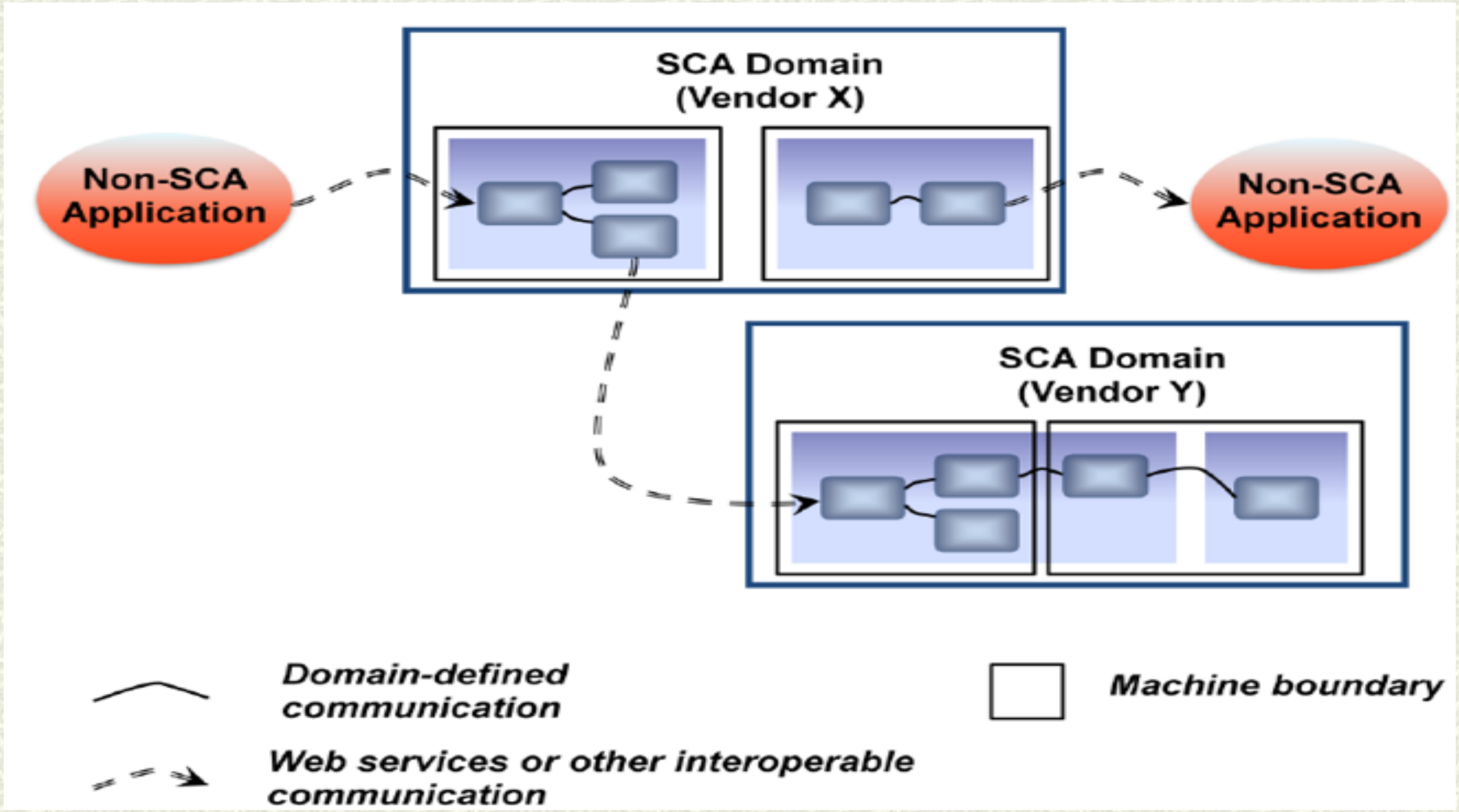
SCA –Domain

- Components and composites are the fundamental elements of every SCA application. Both are contained within a larger construct called a *domain*.
- A domain can contain one or more composites, each of which has components implemented in one or more processes running on one or more machines.
- How communication happens between these components, whether it is intra-process, inter-process, or inter-machine, can be defined differently by each SCA vendor. Whatever choice is made, composites do not span domain boundaries.
- Even though an SCA composite runs in a single-vendor environment, it can still communicate with applications outside its own domain. To do this, an SCA component can make itself accessible using an interoperable protocol such as Web services.
- To communicate between domains, or with non-SCA applications, a component will typically allow access via Web services or some other interoperable mechanism. An SCA application communicating with another SCA application in a different domain sees that application just like a non-SCA application; its use of SCA is not visible outside its domain.

SCA –Domain


SCA Domain
SCA Composite
SCA Component
SCA Component
SCA Component
SCA Component
SCA Component
SCA Composite
SCA Component
SCA Composite
SCA Component
SCA Component
SCA Component
Process boundary
Machine boundary

SCA –Domain

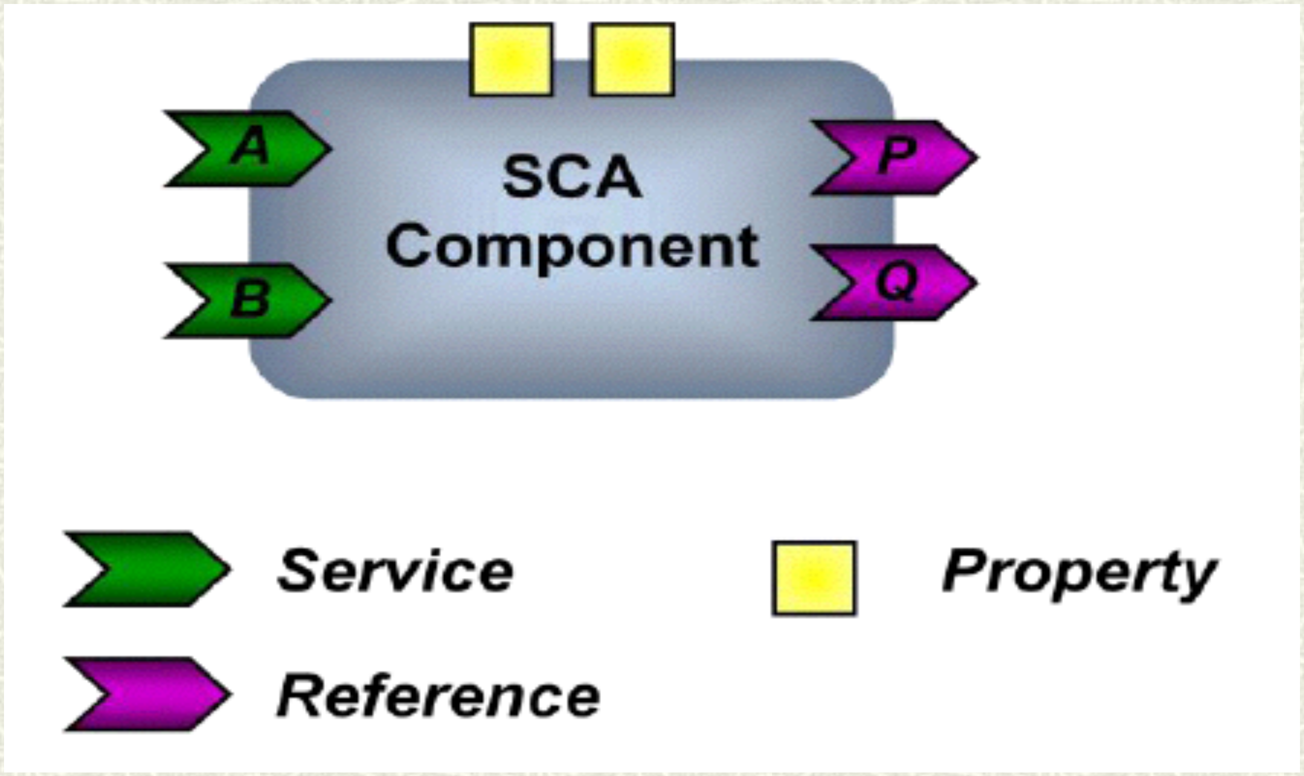

SCA Domain (Vendor X)
Non-SCA Application
Non-SCA Application
SCA Domain (Vendor Y)
Domain-defined communication
Machine boundary
Web services or other interoperable communication

SCA – Component Structure

- Every component relies on a common set of abstractions, including services, references, properties, and bindings, to specify its interactions with the world outside itself.
- A *service* provides some number of operations that can be accessed by the component's client. How services are described depends on the technology used to implement the component (Java interfaces, WSDL for BPEL).
- A component may also rely on services provided by other components in its domain or by software outside its domain. To describe this, a component can indicate the services it relies on using *references*. Each reference defines an interface containing operations that this component needs to invoke.
- Explicitly defining references allow what is known as dependency injection. Instead of requiring a developer to write code that locates the service a component depends on, the SCA runtime can locate that service for the developer.

SCA – Component Structure

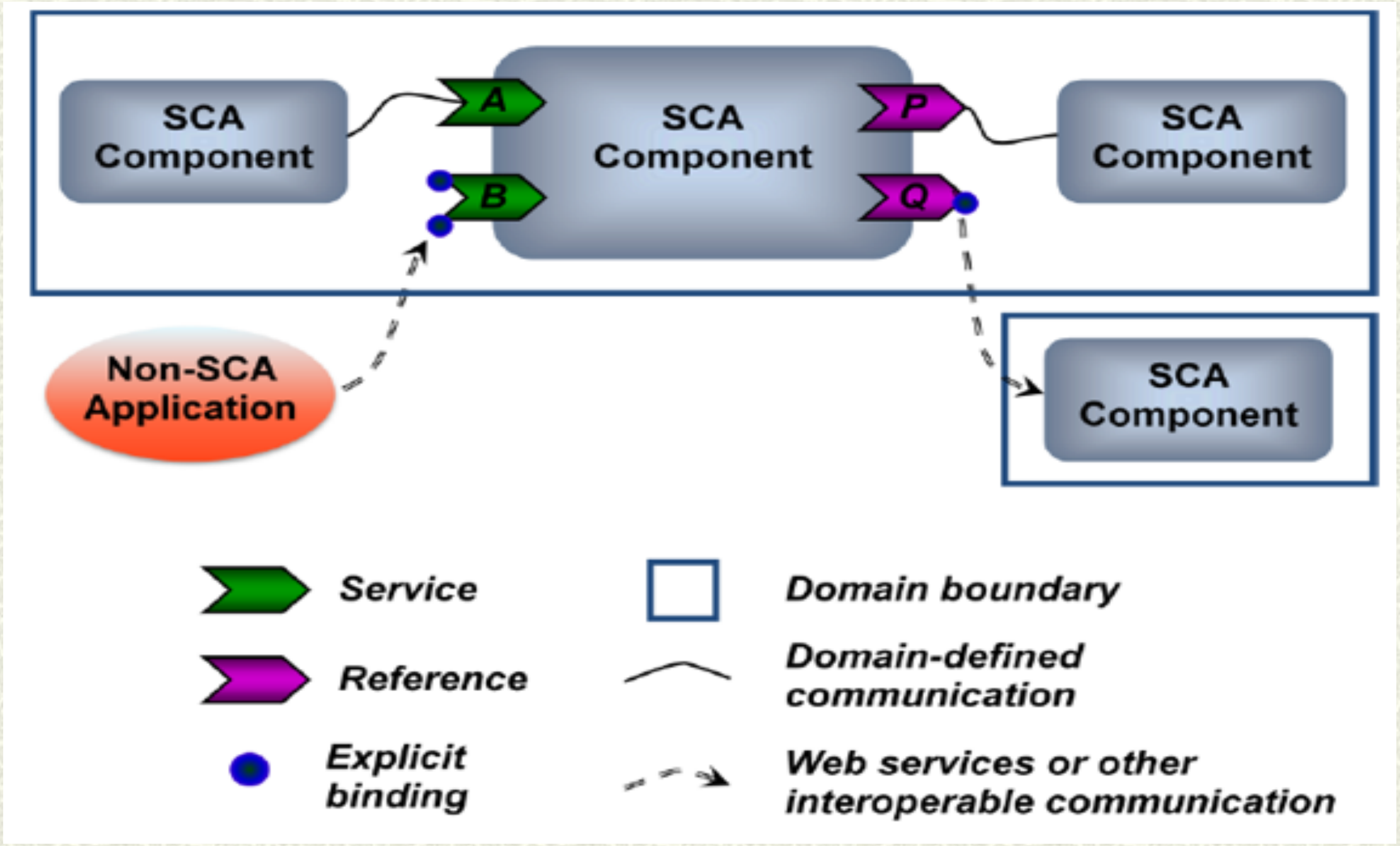
- Along with services and references, a component can also define one or more properties. Each property contains a value that can be read by that component from the SCDL configuration file when it is instantiated.



SCA – Bindings

- Services and references let a component communicate with other software. By design, however, they say nothing about how that communication happens.
- A binding specifies exactly how communication should be done between an SCA component and something else. Depending on what it’s communicating with, a component might or might not have explicitly specified bindings:
  - A component that communicates with another component in the same domain, even one in another process or on another machine, does not need to have any explicit bindings specified. The runtime determines what bindings to use.
  - To communicate outside its domain whether to a non-SCA application or an SCA application running in another domain, a component’s creator (or perhaps the person who deploys the component) must specify one or more bindings for this communication. Each binding defines a particular protocol that can be used to communicate with this service or reference. A single service or reference can have multiple bindings, allowing different remote software to communicate with it in different ways.

SCA – Bindings



SCA Platforms

- **Apache Tuscany** SCA is implemented in Java and C++
- **Fabric3** is a platform for developing, assembling, and managing distributed applications.