The roots of SOA (comparing SOA to past architectures)

Ediz Şaykol

What is architecture?

- **Application architecture**
  - Application architecture is to an application development team what a blueprint is to a team of construction workers.
  - Different organizations document different levels of application architecture. Some keep it high-level, providing abstract physical and logical representations of the technical blueprint. Others include more detail, such as common data models, communication flow diagrams, application-wide security requirements, and aspects of infrastructure.

What is architecture?

- **Enterprise architecture**
  - In larger IT environments, the need to control and direct IT infrastructure is critical. When numerous, disparate application architectures co-exist and sometimes even integrate, the demands on the underlying hosting platforms can be complex and onerous.
  - Therefore, it is common for a master specification to be created, providing a high-level overview of all forms of heterogeneity that exist within an enterprise, as well as a definition of the supporting infrastructure.

Application vs Enterprise

- the relationship between an urban plan and the blueprint of a building are comparable to that of enterprise and application architecture specifications
- changes to enterprise architectures directly affect application architectures
- enterprise architectures often contain a long-term vision of how the organization plans to evolve its technology and environments

Service-oriented architecture

- Service-oriented architecture spans both enterprise and application architecture domains.
- The benefit potential offered by SOA can only be truly realized when applied across multiple solution environments.
  - This is where the investment in building reusable and interoperable services based on a vendor-neutral communications platform can fully be leveraged.
- This does not mean that the entire enterprise must become service-oriented.
  - SOA belongs in those areas that have the most to gain from the features and characteristics it introduces.

SOA vs. client-server architecture

- any environment in which one piece of software requests or receives information from another can be referred to as "client-server.“
- the industry term "client-server architecture" generally refers to a particular generation of early environments during which the client and the server played specific roles and had distinct implementation characteristics

The original monolithic mainframe systems that empowered organizations to get seriously computerized often are considered the first inception of client-server architecture.
These environments, in which bulky mainframe back-ends served thin clients, are considered an implementation of the single-tier client-server architecture

Mainframe systems natively supported both synchronous and asynchronous communication. The latter approach was used primarily to allow the server to continuously receive characters from the terminal in response to individual key-strokes. Only upon certain conditions would the server actually respond.

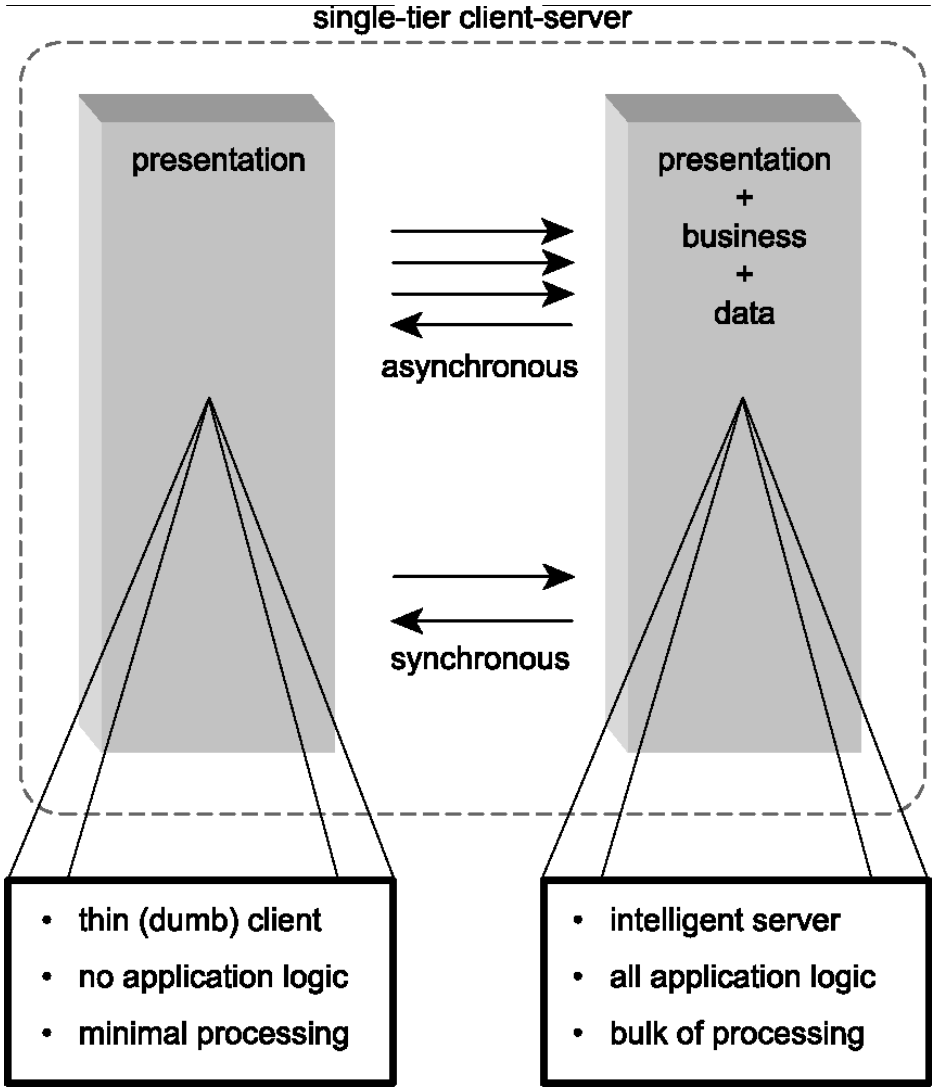

Figure 4.2: A typical single-tier client-server architecture.

new approach introduced the concept of delegating logic and processing duties onto individual workstations, resulting in the birth of the fat client. Further supported by the innovation of the graphical user-interface (GUI), two-tier client-server was considered a huge step forward and went on to dominate the IT world for years during the early 90s.

The common configuration of this architecture consisted of multiple fat clients, each with its own connection to a database on a central server. Client-side software performed the bulk of the processing, including all presentation-related and most data access logic

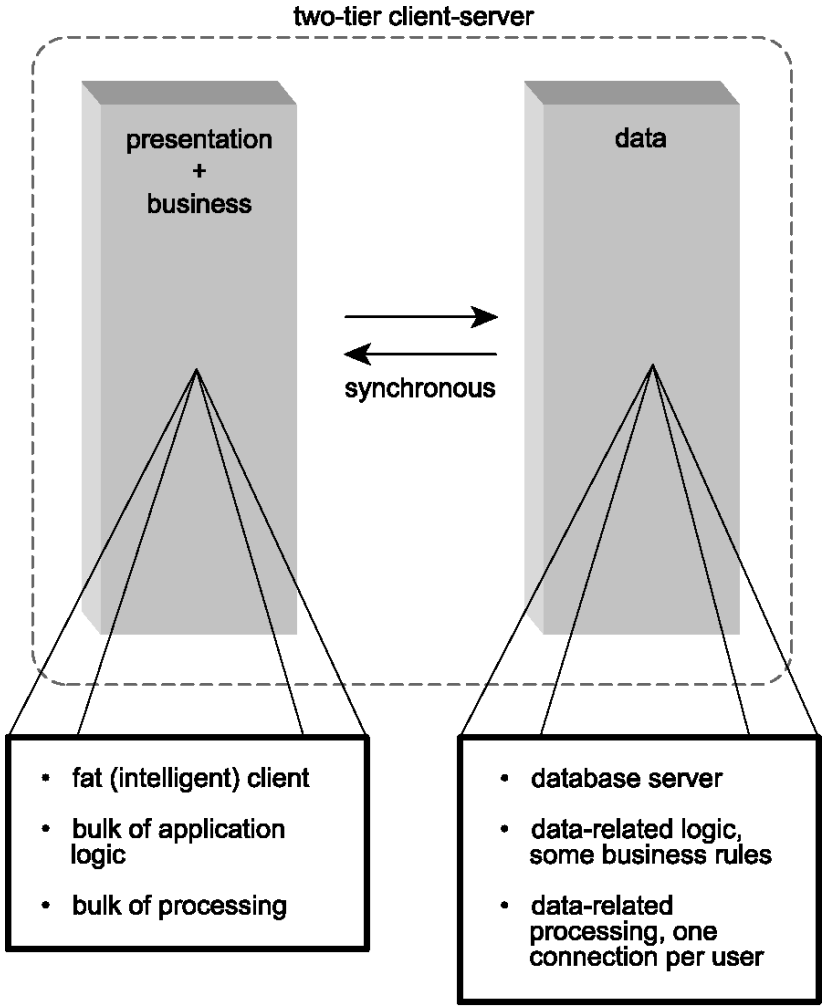

Figure 4.3: A typical two-tier client-server architecture.

Vs. Application Logic

- Client-server environments place the majority of application logic into the client software.
- This results in a monolithic executable that controls the user experience, as well as the back-end resources.

- The presentation layer within contemporary service-oriented solutions can vary. Any piece of software capable of exchanging SOAP messages according to required service contracts can be classified as a service requestor. While it is commonly expected for requestors to be services as well, presentation layer designs are completely open and specific to a solution's requirements.

- Within the server environment, options exist as to where application logic can reside and how it can be distributed. These options do not preclude the use of database triggers or stored procedures.

- the partitioning of processing logic into autonomous units. This facilitates specific design qualities, such as service statelessness and interoperability, as well as future composability and reusability.

In response to the costs and limitations associated with the two-tier client server architecture, the concept of building component-based applications hit the mainstream.

Multi-tier client-server architectures surfaced, breaking up the monolithic client executable into components designed to varying extents of compliance with object-orientation.

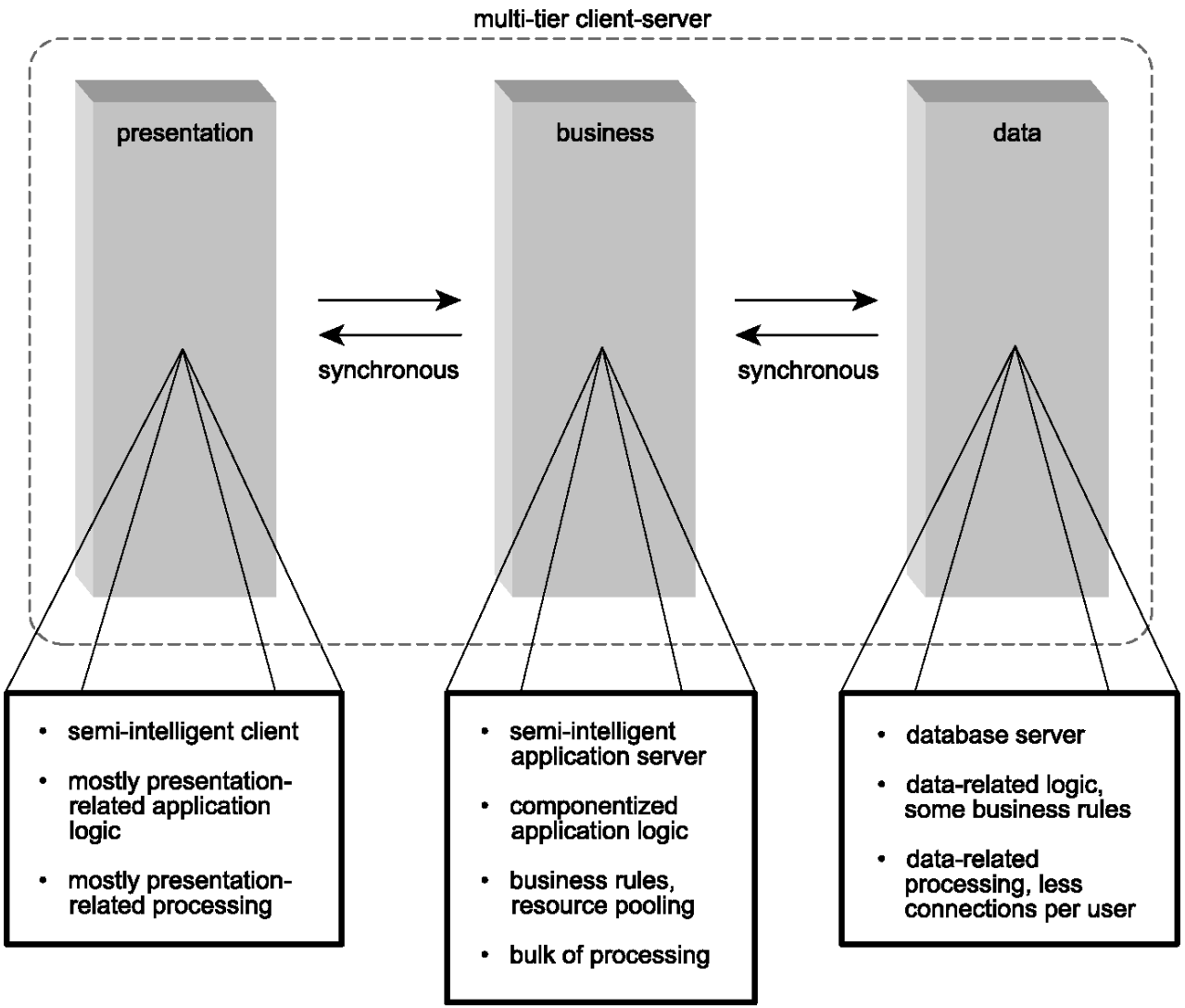

Figure 4.4: A typical multi-tier client-server architecture.

Multi-tier issues

- These benefits came at the cost of increased complexity and ended up shifting expense and effort from deployment issues to development and administration processes.
- Building components capable of processing multiple, concurrent requests was more difficult and problem-ridden than developing a straight-forward executable intended for a single user.
- Additionally, replacing client-server database connections was the client-server remote procedure call (RPC) connection. RPC technologies such as CORBA and DCOM allowed for remote communication between components residing on client workstations and servers.

Upon the arrival of the World Wide Web as a viable medium for computing technology in the mid-to-late 90s, the multi-tiered client-server environments began incorporating Internet technology.

From the late 90s to the mid 2000s, distributed Internet architectures represented the de facto computing platform for custom developed enterprise solutions. The commoditization of component-based programming skills and the increasing sophistication of middleware eventually lessened some of the overall complexity.

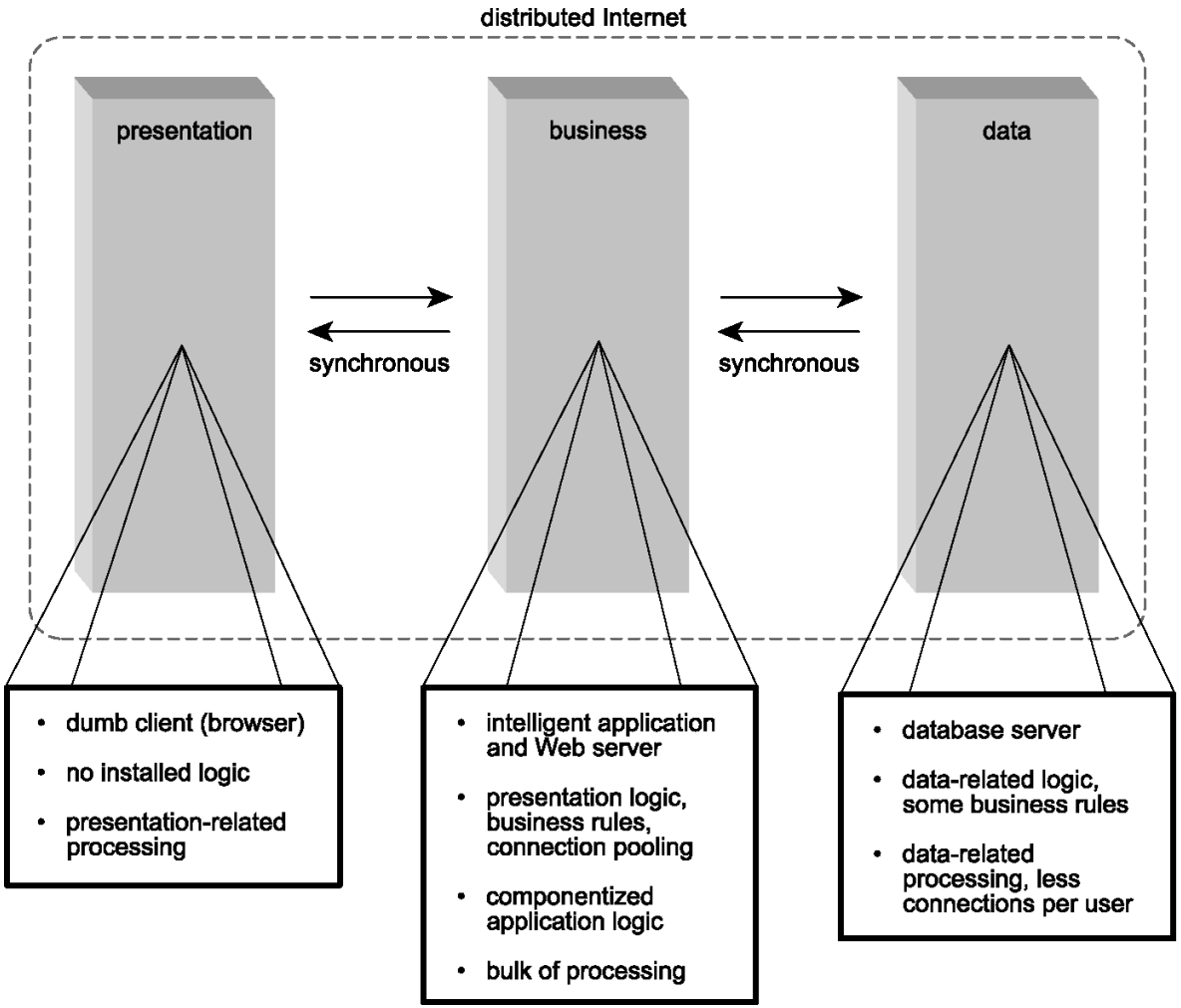

Figure 4.5: A typical distributed Internet architecture.

Distributed on Internet

- Most significant was the replacement of the custom software client component with the browser. Not only did this change radically alter (and limit) user-interface design, it practically shifted 100% of application logic to the server

- Distributed Internet architecture also introduced a new physical tier, the Web server. This resulted in HTTP replacing proprietary RPC protocols used to communicate between the user's workstation and the server. The role of RPC was limited to enabling communication between remote Web and application servers.

App Logic on Web vs SOA

- Except for some rare applications that embed proprietary extensions in browsers, distributed Internet applications place all of their application logic on the server side.
- Even client-side scripts intended to execute in response to events on a Web page are downloaded from the Web server upon the initial HTTP request.
- With none of the logic existing on the client workstation, the entire solution is centralized.
- The emphasis is therefore on:
  - how application logic should be partitioned
  - where the partitioned units of processing logic should reside
  - how the units of processing logic should interact

- From a physical perspective, service-oriented architecture is very similar to distributed Internet architecture. Provider logic resides on the server end where it is broken down into separate units. The differences lie in the principles used to determine the three primary design considerations just listed.

Traditional distributed applications consist of a series of components that reside on one or more application servers. Components are designed with varying degrees of functional granularity, depending on the tasks they execute, and to what extent they are considered reusable by other tasks or applications.
Components residing on the same server communicate via proprietary APIs, as per the public interfaces they expose. RPC protocols are used to accomplish the same communication across server boundaries. This is made possible through the use of local proxy stubs that represent components in remote locations

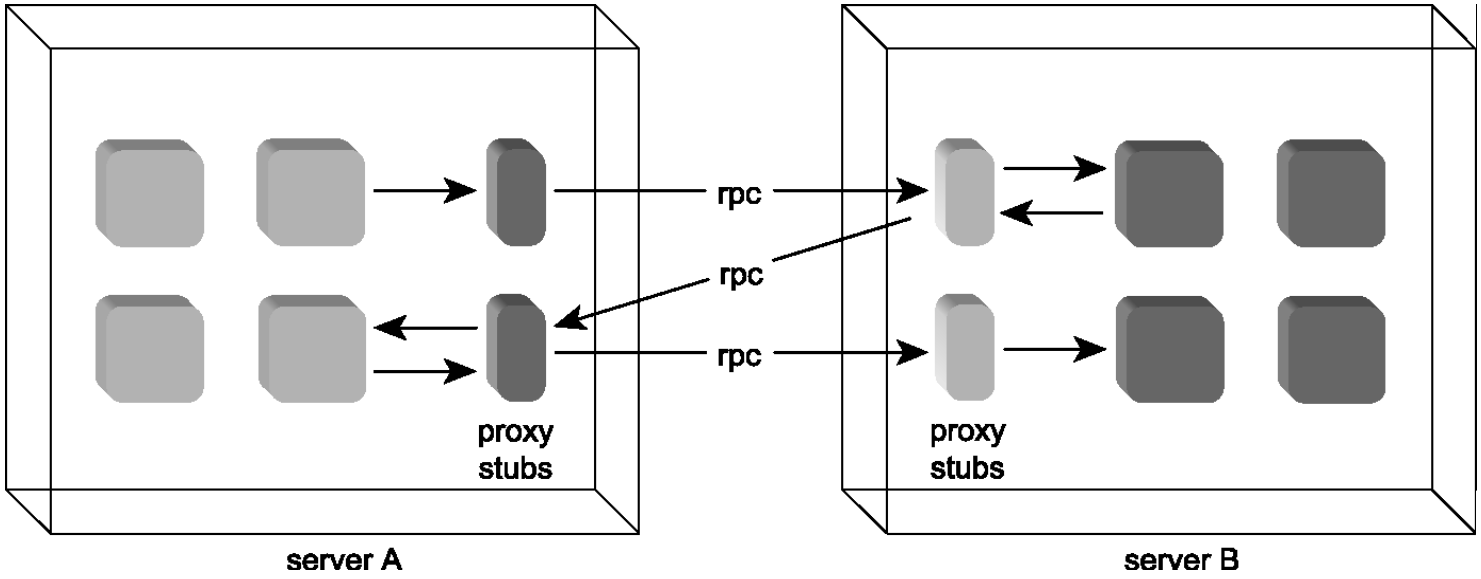

Figure 4.6: Components rely on proxy stubs for remote communication.

The primary role of Web services in this context has been to introduce an integration layer that consists of wrapper services that enable synchronous communication via SOAP-compliant integration channels

the initial release of the SOAP specification and the first generation of SOAP servers were specifically designed to duplicate RPC-style communication using messages.

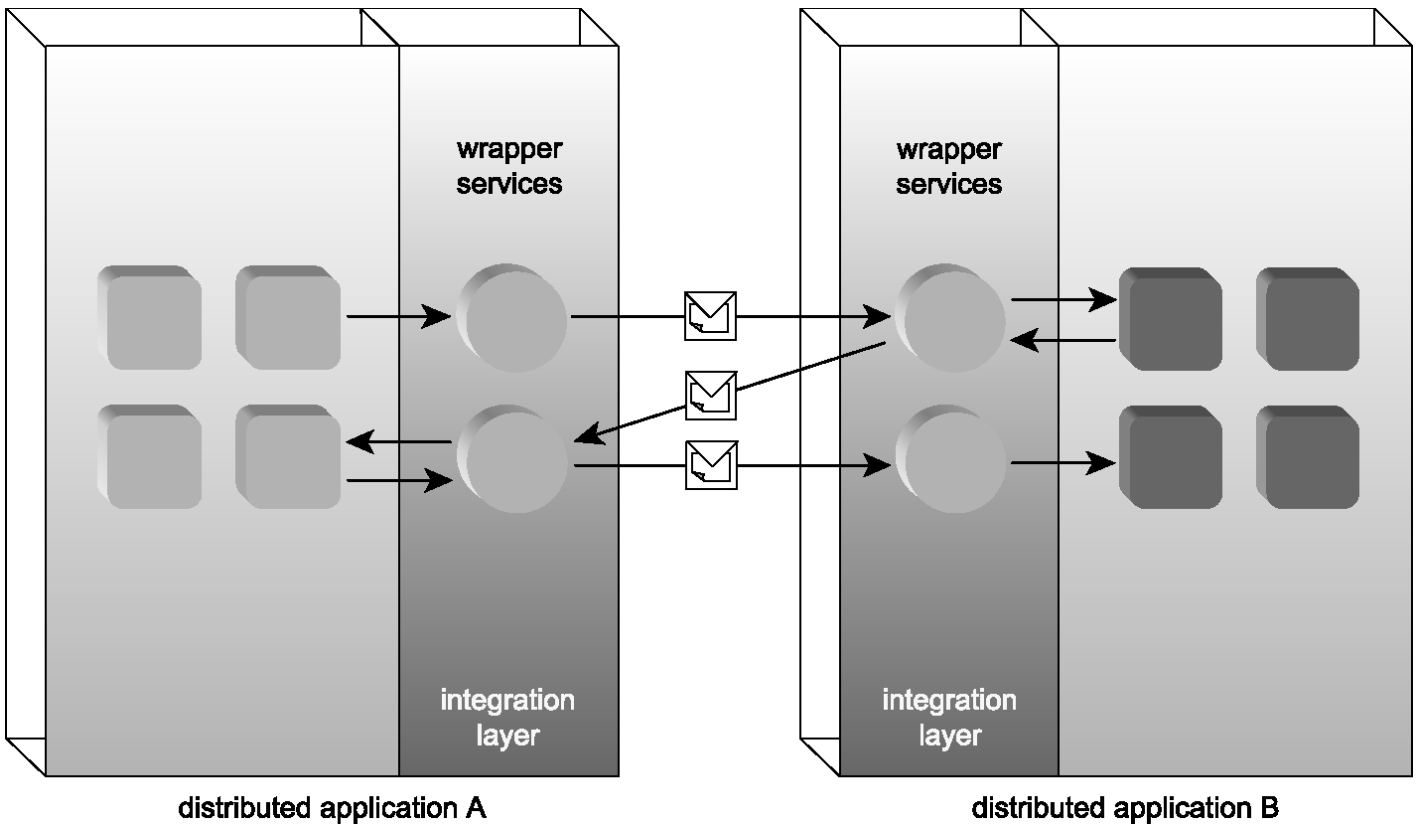

Figure 4.7: Wrapper services encapsulating components.

Wrapper Web Services with SOAP

- These integration channels are primarily utilized in integration architectures to facilitate communication with other applications or outside partners.
- They also are used to enable communication with other (more service-oriented) solutions and to take advantage of some of the features offered by third-party utility Web services.
- Regardless of their use or purpose within traditional architectures, it is important to clarify that a distributed Internet architecture that incorporates Web services in this manner does not qualify as a true SOA.
- It is simply a distributed Internet architecture that uses Web services.

Service-orientation and object-orientation

- Service-orientation emphasizes loose coupling between units of processing logic (services). Although object-orientation supports the creation of reusable, loosely coupled programming routines, much of it is based on predefined class dependencies, resulting in more tightly bound units of processing logic (objects).
- Service-orientation encourages coarse-grained interfaces (service descriptions) so that every unit of communication (message) contains as much information as possible for the completion of a given task. Object-oriented programming fully supports fine-grained interfaces (APIs) so that units of communication (RPC or local API calls) can perform various sized tasks.
- Service-orientation expects the scope of a unit of processing logic (service) to vary significantly. Object-oriented units of logic (objects) tend to be smaller and more specific in scope.

Service-orientation and object-orientation

- Service-orientation promotes the creation of activity-agnostic units of processing logic (services) that are driven into action by intelligent units of communication (messages). Object-orientation encourages the binding of processing logic with data, resulting in highly intelligent units (objects).
- Service-orientation prefers that units of processing logic (services) be designed to remain as stateless as possible. Object-orientation promotes the binding of data and logic, resulting in the creation of more stateful units (objects). (However, more recent component-based design approaches deviate from this tendency.)
- Service-orientation supports the composition of loosely coupled units of processing logic (services). Object-orientation supports composition but also encourages inheritance among units of processing logic (objects), which can lead to tightly coupled dependencies.