Space-Based Architecture

The space-based architecture pattern is specifically designed to address and solve scalability and concurrency issues. It is also a useful architecture pattern for applications that have variable and unpredictable concurrent user volumes. Solving the extreme and variable scalability issue *architecturally* is often a better approach than trying to scale out a database or retrofit caching technologies into a non-scalable architecture.

The space-based pattern (also sometimes referred to as the cloud architecture pattern) minimizes the factors that limit application scaling. This pattern gets its name from the concept of *tuple space,* the idea of distributed shared memory. High scalability is achieved by removing the central database constraint and using replicated in-memory data grids instead. Application data is kept in-memory and replicated among all the active processing units. Processing units can be dynamically started up and shut down as user load increases and decreases, thereby addressing variable scalability. Because there is no central database, the database bottleneck is removed, providing near-infinite scalability within the application.

Most applications that fit into this pattern are standard websites that receive a request from a browser and perform some sort of action. A bidding auction site is a good example of this. The site continually receives bids from internet users through a browser request. The application would receive a bid for a particular item, record that bid with a timestamp, and update the latest bid information for the item, and send the information back to the browser.

1. Components

There are two primary components within this architecture pattern: a *processing unit* and *virtualized middleware*. Below figure (*Figure 1: Space-based architecture pattern*) illustrates the basic space-based architecture pattern and its primary architecture components.

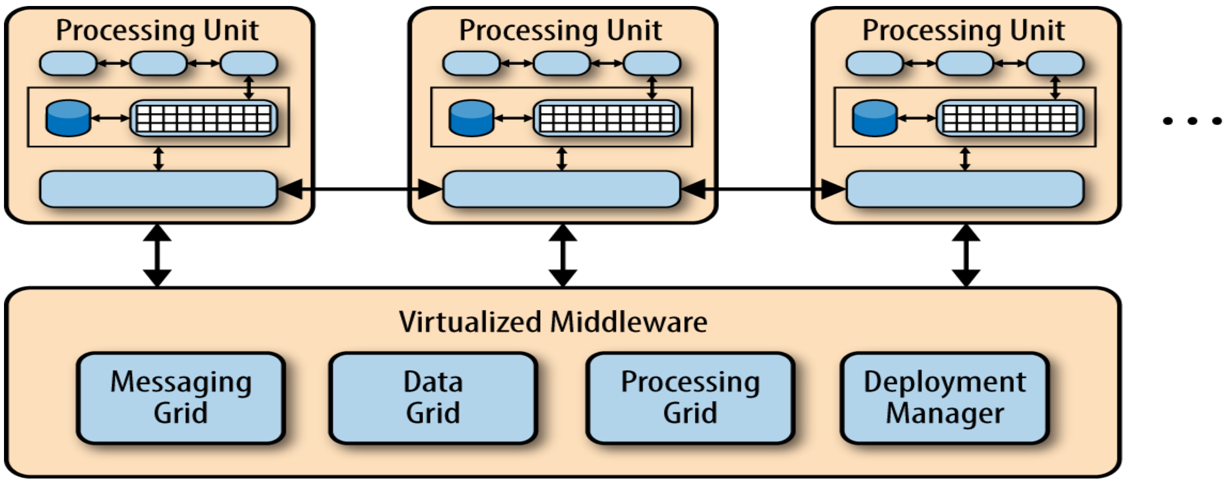

Figure 1: Space-based architecture pattern

1.1. Processing-Unit

The processing-unit component contains the application components (or portions of the application components). This includes web-based components as well as backend business logic. The contents of the processing unit vary based on the type of application—smaller web-based applications would likely be deployed into a single processing unit, whereas larger applications may split the application functionality into multiple processing units based on the functional areas of the application. The processing unit typically contains the application modules, along with an in-memory data grid and an optional asynchronous persistent store for failover. It also contains a replication engine that is used by the virtualized middleware to replicate data changes made by one processing unit to other active processing units.

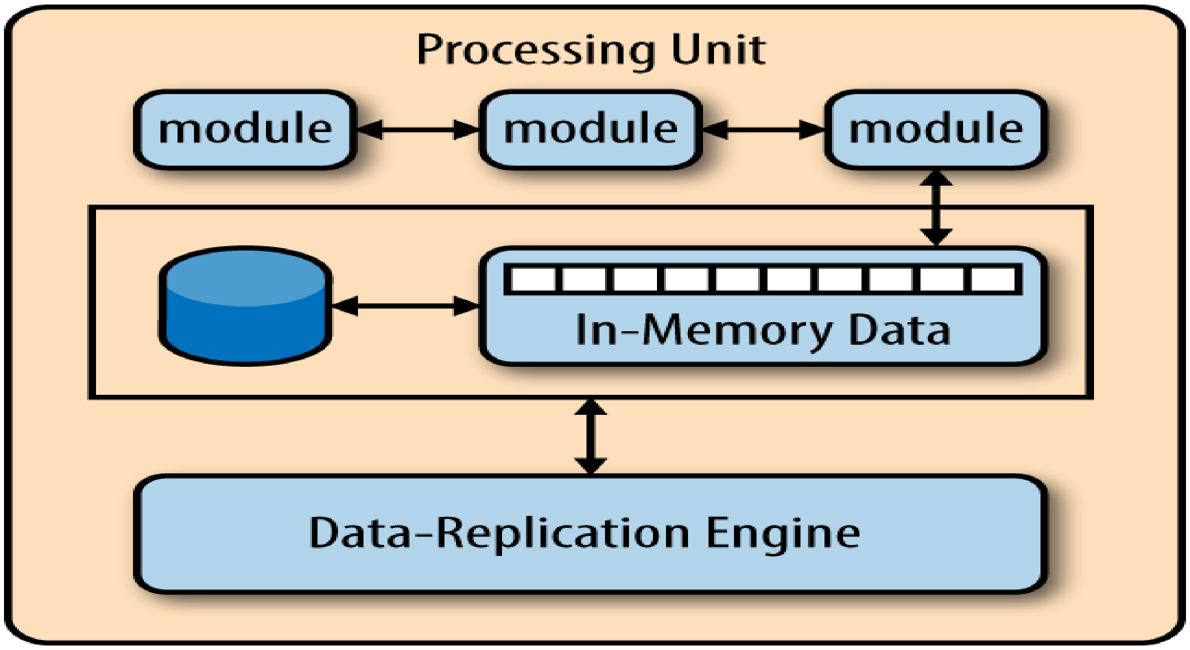

Figure 2: Processing-unit component

The magic of the space-based architecture pattern lies in the virtualized middleware components and the in-memory data grid contained within each processing unit. Above figure (*Figure 2: Processing-unit component*) shows the typical processing unit architecture containing the application modules, in-memory data grid, optional asynchronous persistence store for failover, and the data-replication engine.

1.2. Virtualized Middleware

The virtualized middleware is essentially the controller for the architecture and manages requests, sessions, data replication, distributed request processing, and process-unit deployment. There are four main architecture components in the virtualized middleware: the messaging grid, the data grid, the processing grid, and the deployment manager.

1.2.1. Messaging Grid

The messaging grid, shown in below figure (*Figure 3: Messaging-grid*), manages input request and session information. When a request comes into the virtualized-middleware component, the messaging-grid component determines which active processing components are available to receive the request and forwards the request to one of those processing units. The complexity of the messaging grid can range from a simple round-robin algorithm to a more complex next-available algorithm that keeps track of which request is being processed by which processing unit.

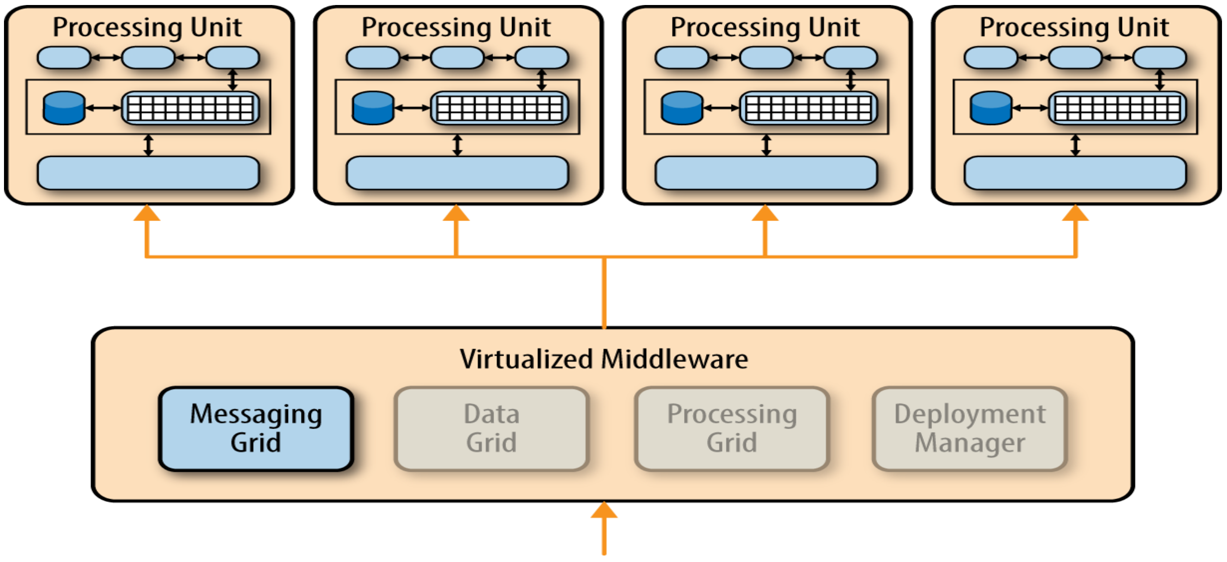

Figure 3: Messaging-grid

1.2.2. Data Grid

The data-grid component is perhaps the most important and crucial component in this pattern. The data grid interacts with the data-replication engine in each processing unit to manage the data replication between processing units when data updates occur.

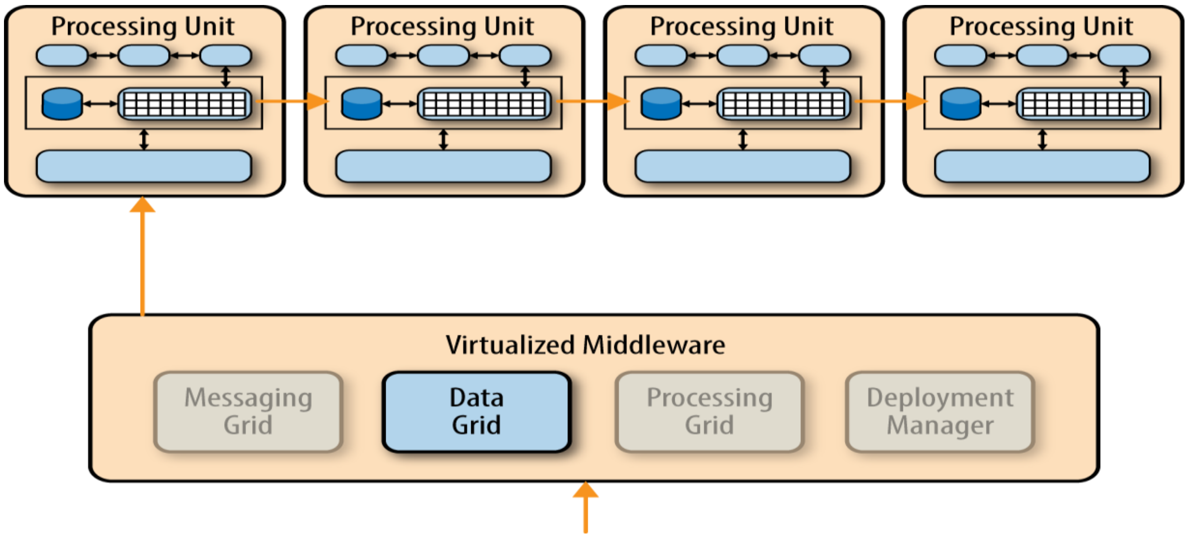

Figure 4: Data-grid

Since the messaging grid can forward a request to any of the processing units available, it is essential that each processing unit contains *exactly* the same data in its in-memory data grid. Although above figure (*Figure 4: Data-grid*) shows a synchronous data replication between processing units, in reality this is done in parallel asynchronously and *very quickly*, sometimes completing the data synchronization in a matter of microseconds (one millionth of a second).

1.2.3. Processing Grid

The processing grid, illustrated in below figure (*Figure 5: Processing grid*), is an optional component within the virtualized middleware that manages distributed request processing when there are multiple processing units, each handling a portion of the application. If a request comes in that requires coordination between processing unit types (e.g., an order processing unit and a customer processing unit), it is the processing grid that mediates and orchestrates the request between those two processing units.

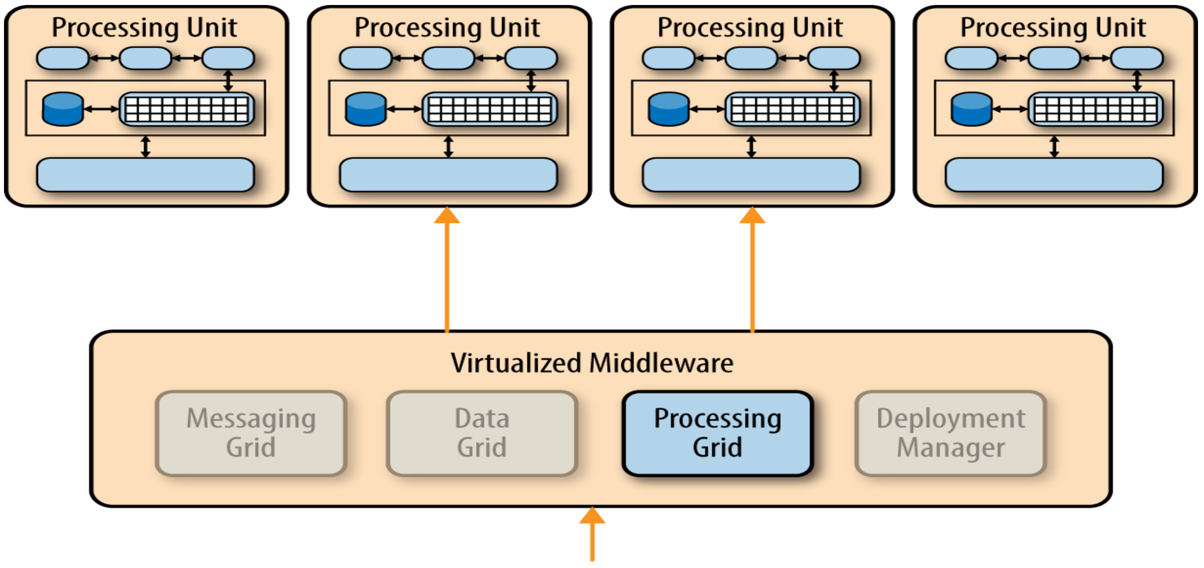

Figure 5: Processing grid

1.2.4. Deployment Manager

The deployment-manager component manages the dynamic startup and shutdown of processing units based on load conditions. This component continually monitors response times and user loads, and starts up new processing units when load increases, and shuts down processing units when the load decreases. It is a critical component to achieving variable scalability needs within an application.

2. Considerations

The space-based architecture pattern is a complex and expensive pattern to implement. It is a good architecture choice for smaller web-based applications with variable load (e.g., social media sites, bidding and auction sites). However, it is not well suited for traditional large-scale relational database applications with large amounts of operational data.

Although the space-based architecture pattern does not require a centralized datastore, one is commonly included to perform the initial in-memory data grid load and asynchronously persist data updates made by the processing units. It is also a common practice to create separate partitions that isolate volatile and widely used transactional data from non-active data, in order to reduce the memory footprint of the in-memory data grid within each processing unit.

It is important to note that while the alternative name of this pattern is the cloud-based architecture, the processing units (as well as the virtualized middleware) do not have to reside on cloud-based hosted services or PaaS (platform as a service). It can just as easily reside on local servers, which is one of the reasons for the name "space-based architecture".

3. Implementation

From a product implementation perspective, we can implement many of the architecture components in this pattern through third-party products such as GemFire, JavaSpaces, GigaSpaces, IBM Object Grid, nCache, and Oracle Coherence. Because the implementation of this pattern varies greatly in terms of cost and capabilities (particularly data replication times), as an architect, you should first establish what your specific goals and needs are before making any product selections.

4. Pattern Analysis

The following table contains a rating and analysis of the common architecture characteristics for the space-based architecture pattern. The rating for each characteristic is based on the natural tendency for that characteristic as a capability based on a typical implementation of the pattern, as well as what the pattern is generally known for.

1.3. Overall agility

Rating: High
Analysis: Overall agility is the ability to respond quickly to a constantly changing environment. Because processing units (deployed instances of the application) can be brought up and down quickly, applications respond well to changes related to an increase or decrease in user load (environment changes). Architectures created using this pattern generally respond well to coding changes due to the small application size and dynamic nature of the pattern.

1.4. Ease of deployment

Rating: High
Analysis: Although space-based architectures are generally not decoupled and distributed, they are dynamic, and sophisticated cloud-based tools allow for applications to easily be “pushed” out to servers, simplifying deployment.

1.5. Testability

Rating: Low

Analysis: Achieving very high user loads in a test environment is both expensive and time consuming, making it difficult to test the scalability aspects of the application.

1.6. Performance

Rating: High
Analysis: High performance is achieved through the in-memory data access and caching mechanisms build into this pattern.

1.7. Scalability

Rating: High
Analysis: High scalability come from the fact that there is little or no dependency on a centralized database, therefore essentially removing this limiting bottleneck from the scalability equation.

1.8. Ease of development

Rating: Low
Analysis: Sophisticated caching and in-memory data grid products make this pattern relatively complex to develop, mostly because of the lack of familiarity with the tools and products used to create this type of architecture. Furthermore, special care must be taken while developing these types of architectures to make sure nothing in the source code impacts performance and scalability.