USING PATTERNS IN ARCHITECTURE OF SOFTWARE ENGINEERING

M.Sc. Ivanova Milka
Faculty of Mechanical Engineering – Technical University of Sofia, Bulgaria
milkachr@tu-sofia.bg

Abstract: *Architecture patterns help define the basic characteristics and behavior of an application. Some architecture patterns lend themselves toward highly scalable applications, whereas other architecture patterns naturally lend themselves toward applications that are highly agile. Knowing the characteristics, strengths, and weaknesses of each architecture pattern is necessary in order to choose the one that meets your specific business needs and goals.As a software architect, you must always justify your architecture decisions, particularly when it comes to choosing a particular architecture pattern or approach. The goal of this report is to give you enough information to make and justify that decision.*

Keywords*: Software engineering, pattern, software engineering architectures, architectural patterns,*

There are five types software engineering architecture where patterns are used

Layered Architecture

The most common architecture pattern is the layered architecture pattern, otherwise known as the n-tier architecture pattern. This pattern is the de facto standard for most EE applications and therefore is widely known by most architects, designers, and developers. The layered architecture pattern closely matches the traditional IT communication and organizational structures found in most companies, making it a natural choice for most business application development efforts.

Event-Driven Architecture

The event-driven architecture pattern is a popular distributed asynchronous architecture pattern used to produce highly scalable applications. It is also highly adaptable and can be used for small applications and as well as large, complex ones. The event-driven architecture is made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events. The event-driven architecture pattern consists of two main topologies, the mediator and the broker. The mediator topology is commonly used when you need to orchestrate multiple steps within an event through a central mediator, whereas the broker topology is used when you want to chain events together without the use of a central mediator. Because the architecture characteristics and implementation strategies differ between these two topologies, it is important to understand each one to know which is best suited for your particular situation.

Microkernel Architecture

The microkernel architecture pattern (sometimes referred to as the plug-in architecture pattern) is a natural pattern for implementing product-based applications. A product-based application is one that is packaged and made available for download in versions as a typical third-party product. However, many companies also develop and release their internal business applications like software products, complete with versions, release notes, and pluggable features. These are also a natural fit for this pattern. The microkernel architecture pattern allows you to add additional application features as plug-ins to the core application, providing extensibility as well as feature separation and isolation.

Microservices Architecture Pattern

The microservices architecture pattern is quickly gaining ground in the industry as a viable alternative to monolithic applications and service-oriented architectures. Because this architecture pattern is still evolving, there's a lot of confusion in the industry about what this pattern is all about and how it is implemented. This section of the report will provide you with the key concepts and foundational knowledge necessary to understand the benefits (and trade-offs) of this important architecture pattern and whether it is the right pattern for your application.

Space based architecture

Most web-based business applications follow the same general request flow: a request from a browser hits the web server, then an application server, then finally the database server. While this pattern works great for a small set of users, bottlenecks start appearing as the user load increases, first at the web-server layer, then at the application-server layer, and finally at the database-server layer. The usual response to bottlenecks based on an increase in user load is to scale out the web servers. This is relatively easy and inexpensive, and sometimes works to address the bottleneck issues. However, in most cases of high user load, scaling out the web-server layer just moves the bottleneck down to the application server. Scaling application servers can be more complex and expensive than web servers and usually just moves the bottleneck down to the database server, which is even more difficult and expensive to scale. Even if you can scale the database, what you eventually end up with is a triangle-shaped topology, with the widest part of the triangle being the web servers and the smallest part being the database

In any high-volume application with an extremely large concurrent user load, the database will usually be the final limiting factor in how many transactions you can process concurrently. While various caching technologies and database scaling products help to address these issues, the fact remains that scaling out a normal application for extreme loads is a very difficult proposition.

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.

Pattern Description

The space-based 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 inmemory 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.

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

The processing-unit component contains the application components. This includes web-based components as well as backend business logic. The contents of the processing unit varies 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.

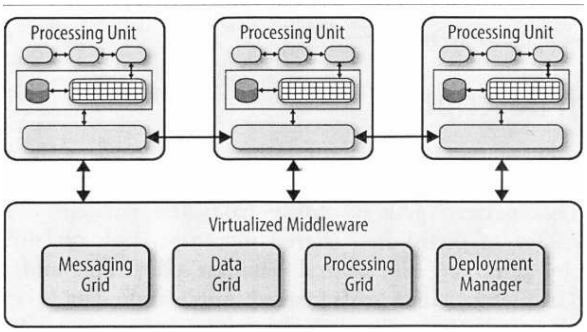

Figure 1. Space-based architecture pattern

The virtualized-middleware component handles housekeeping and communications. It contains components that control various aspects of data synchronization and request handling. Included in the virtualized middleware are the messaging grid, data grid, processing grid, and deployment manager. These components, which are described in detail in the next section, can be custom written or purchased as third-party products.

Pattern Dynamics

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. Figure 2 shows the typical processing unit architecture containing the application modules, inmemory data grid, optional asynchronous persistence store for failover, and the data-replication engine.

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.

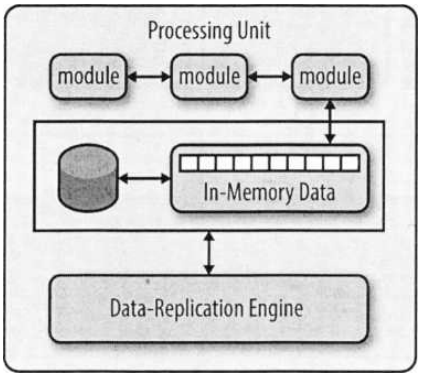

Figure 2. Processing-unit component

Messaging Grid

The messaging grid, shown in Figure 3, 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.

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. 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.

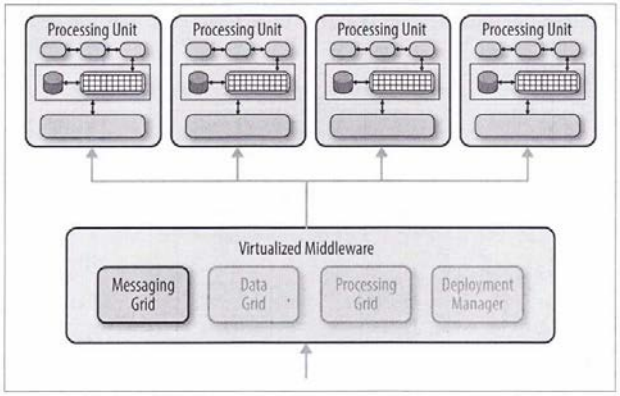

Figure 3 Processing-grid component

Although Figure 4 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).

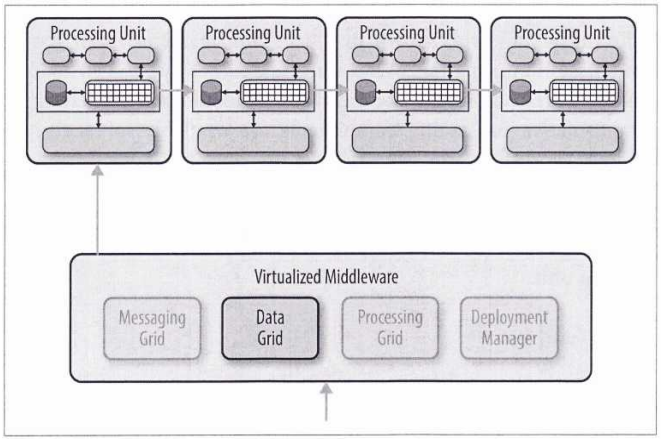

Figure 4. Processing-unit component

Processing Grid

The processing grid, illustrated in Figure 5, 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.

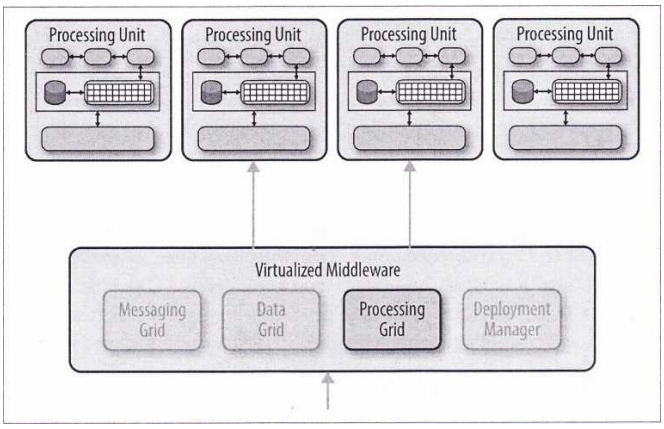

Figure 5. Processing-grid component

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.

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 data store, 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 I prefer the name "space-based architecture."

From a product implementation perspective, you can implement many of the architecture components in this pattern through third- party products. 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.

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. For a side-by-side comparison of how this pattern relates to other patterns in this report, please refer to Appendix A at the end of this report.

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.

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.

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.

Performance

Rating: High

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

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.

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.

Summary

Table A-l summarizes the pattern-analysis scoring for each of the architecture patterns described in this report. This summary will help you determine which pattern might be best for your situation. For example, if your primary architectural concern is scalability, you can look across this chart and see that the event-driven pattern, microservices pattern, and space-based pattern are probably good architecture pattern choices. Similarly, if you choose the layered architecture pattern for your application, you can refer to the chart to see that deployment, performance, and scalability might be risk areas in your architecture

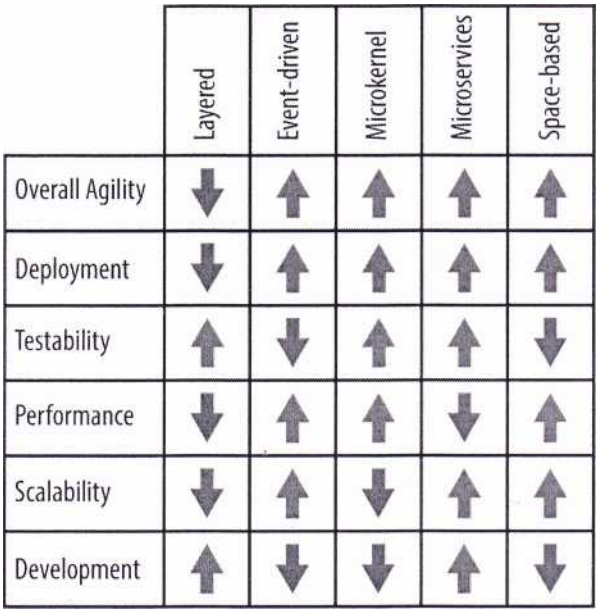

Table 1

While this chart will help guide you in choosing the right pattern, there is much more to consider when choosing an architecture pattern. You must analyze all aspects of your environment, including infrastructure support, developer skill set, project budget, project deadlines, and application size (to name a few). Choosing the right architecture pattern is critical, because once an architecture is in place, it is very hard (and expensive) to change.

Bibliography:

1. O'Reilly, Software Architecture Patterns, Mark Richards'2016