Chapter 16. Space-Based Architecture Style

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 typical request flow works for a small set of concurrent users, bottlenecks start appearing as the concurrent 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 when increased user load causes bottlenecks is to scale out the web servers. This is relatively easy and inexpensive, and sometimes it works. 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 scaling web servers, and doing so 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 (easiest to scale) and the smallest part being the database (hardest to scale), as illustrated in Figure 16-1.

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

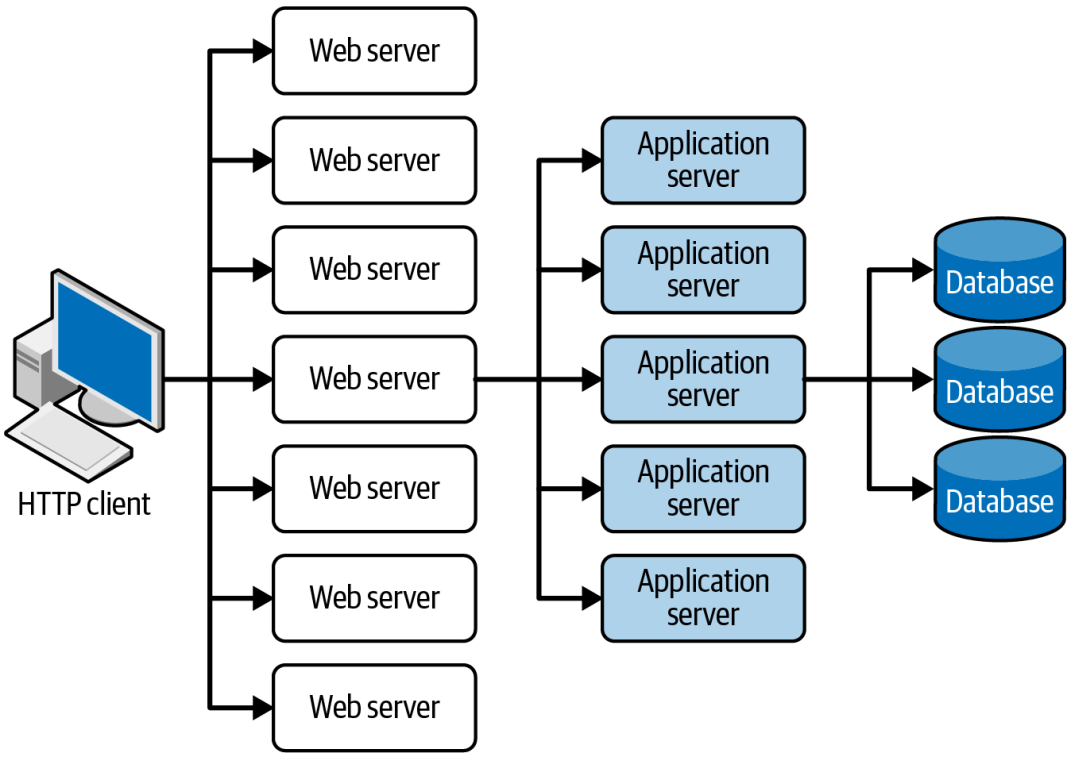

Figure 16-1. Scalability limits within a traditional web-based topology

The *space-based* architecture style is specifically designed to address problems involving high scalability, elasticity, and concurrency. It is also a useful architecture style for applications that have variable and unpredictable concurrent user volumes. It's often better to solve extreme and variable scalability architecturally, rather than try to scale out a database or retrofit caching technologies into an architecture that can't scale as well.

Topology

Space-based architecture gets its name from the concept of *tuple space*, the technique of using multiple parallel processors that communicate through shared memory. Space-based systems achieve high scalability, elasticity, and performance by replacing the central database as a synchronous constraint in the system with replicated in-memory data grids.

Application data is kept in-memory and replicated among all the active processing units. When a processing unit updates data, it asynchronously sends that data to the database through a data pump, usually via messaging with persistent queues. Processing units start up and shut down dynamically as user load increases and decreases, which addresses variable scalability. Because no central database is involved in the application's standard transactional processing, the database bottleneck is removed. This allows for near-infinite scalability within the application.

Space-based architecture is a complicated architectural style consisting of many different artifacts that work together to process a single request. Its primary artifacts are:

Processing units

: Contain the application functionality

Virtualized middleware

: The collection of infrastructure-related artifacts that are used to manage and coordinate the processing units

Messaging grid

: Used to manage input requests and session state

Data grid

: Manages the synchronization and replication of data between processing units

Processing grid

: Used to manage request orchestration between multiple processing units

Deployment manager

: Manages the starting and tearing down of processing unit instances as load increases and decreases

Data pumps

: Asynchronously send updated data to the database

Data writers

Perform the updates from the data pumps

Data readers

Read database data and deliver it to processing units upon startup

Figure 16-2 illustrates these primary artifacts.

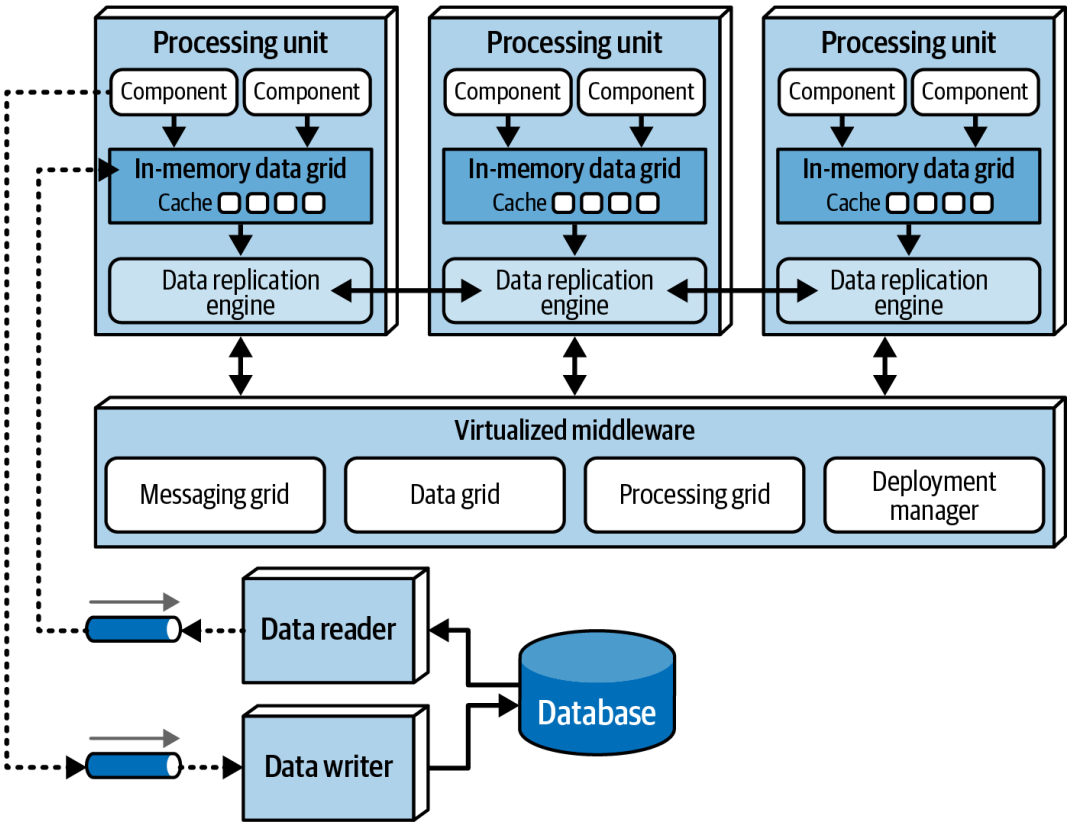

Figure 16-2. Space-based architecture's basic topology

Style Specifics

The following sections describe these primary artifacts and how they work in detail.

Processing Unit

The processing unit (illustrated in Figure 16-3) contains the application logic (or portions of it), usually including 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 are usually deployed into a single processing unit, whereas larger applications often split their functionality into multiple processing units, based on the application's functional areas. The processing unit can also contain small, single-purpose services (much like microservices). In addition, the processing unit also contains an in-memory data grid and replication engine, which are usually implemented through such products as Hazelcast, Apache Ignite, and Oracle Coherence (see "Data Grid").

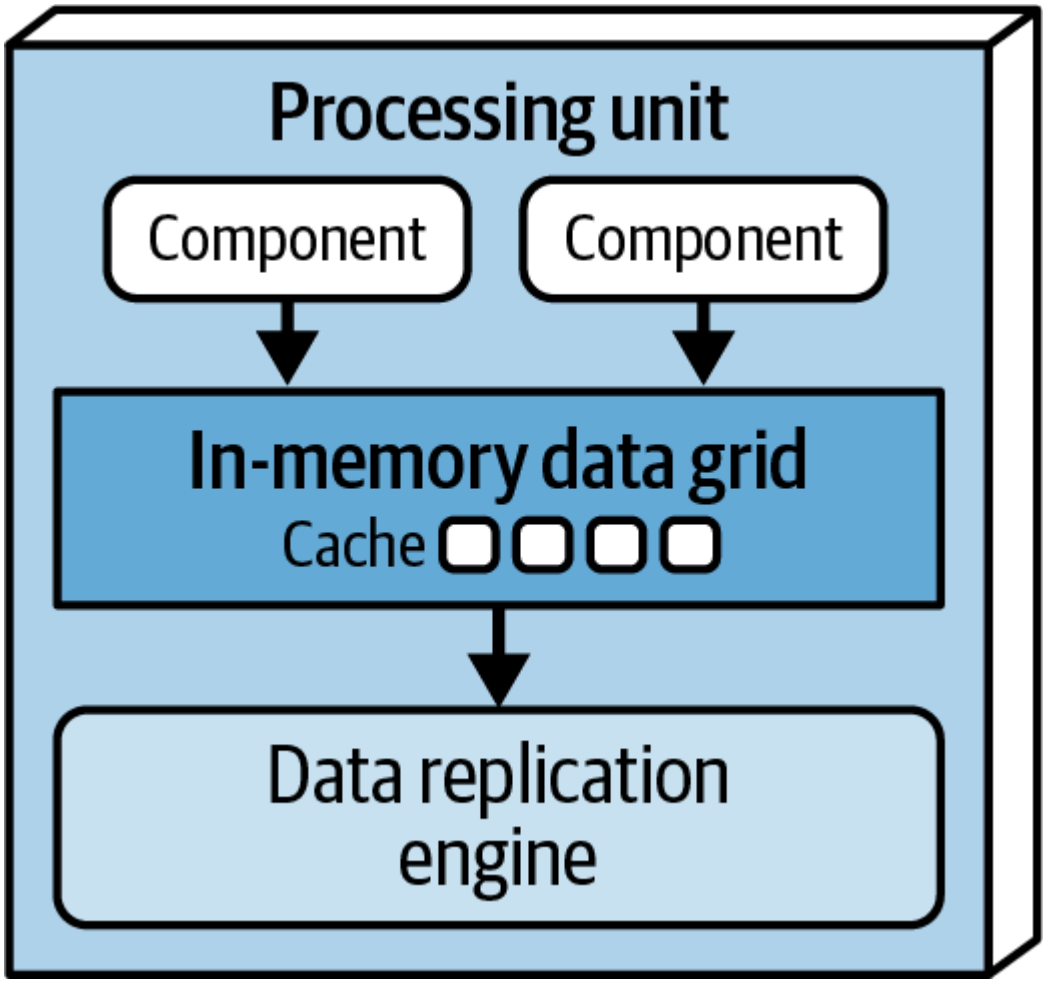

Figure 16-3. The processing unit contains the application's functionality

Virtualized Middleware

The *virtualized middleware*, as shown in Figure 16-4, contains various infrastructure-related artifacts and is used to manage and control the processing units. At a minimum, this middleware artifact contains a *messaging grid* to manage input requests and user session state, a *data grid* to manage data replication and synchronization, and a *deployment manager* to start and tear down processing unit instances as they are needed and not needed. Optionally, the virtualized middleware also contains a *processing grid* in the event two or more processing units need to be orchestrated for a single business request. An architect can add any other infrastructure-related function to the virtualized middleware as needed, such as security functionality, metrics gathering for observability, and so on.

Since no single product can perform all of the functions of the virtualized middleware, it is usually implemented through third-party products such as web servers, caching tools, load balancers, service orchestrators, and deployment managers to manage monitoring, starting, and tearing down the processing units. Each of these middleware artifacts is described in detail in the following sections.

Messaging Grid

The *messaging grid*, shown in Figure 16-4, is part of the virtualized middleware and manages input requests and session state. When a request comes into the virtualized middleware, the messaging grid component determines which active

processing units are available to receive it, then forwards the request to one of those processing units.

The complexity of the messaging grid can vary, from a simple round-robin algorithm to a more complex next-available algorithm that keeps track of which processing unit is most available. This component is usually implemented using a typical web server with load-balancing capabilities (such as HA Proxy or Nginx).

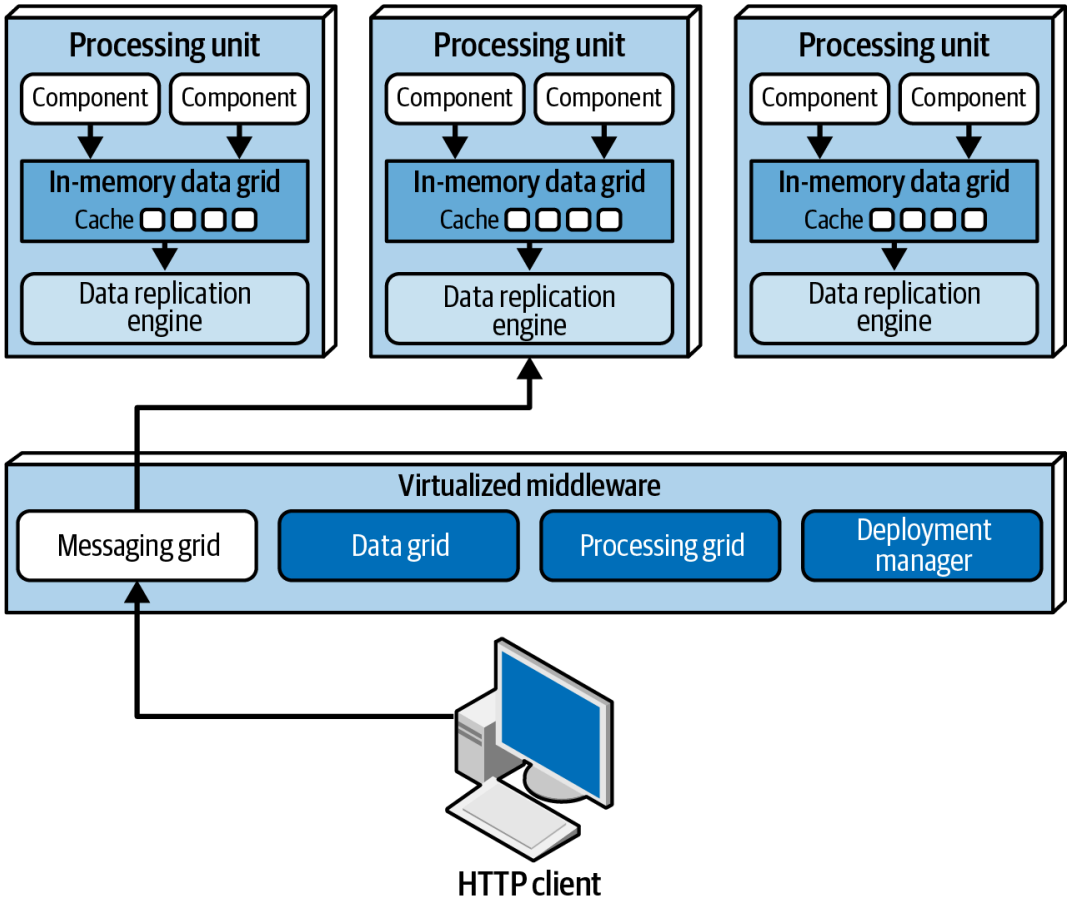

Figure 16-4. The messaging grid handles requests and session state

Data Grid

The *data grid* is a component—perhaps the most crucial component—of the virtualized middleware. In most modern implementations, the data grid is implemented solely within the processing units as an in-memory replicated cache (see "Replicated and distributed caching"). However, for replicated caching implementations that require an external controller, or that use a distributed cache, this functionality resides in *both* the processing units and the data grid components of the virtualized middleware.

Since the messaging grid can forward a request to any of the processing units available, it is essential that each processing unit's in-memory data grid contains *exactly the same data*. As illustrated by the dotted lines in Figure 16-5, data replication is typically done asynchronously between processing units, usually completing data replication in less than 100 ms.

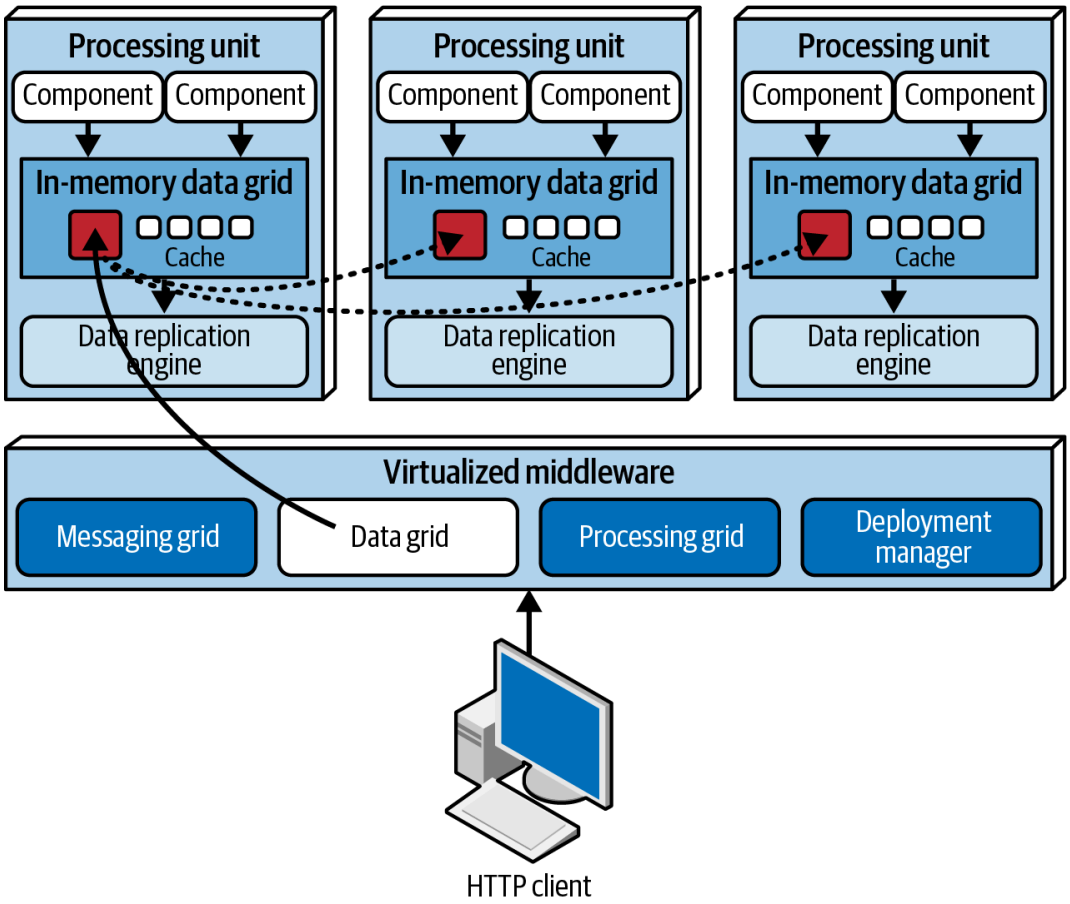

Figure 16-5. The data grid synchronizes the in-memory caches

Data is synchronized between processing units that contain the same named data grid. For example, the following Java code uses Hazelcast to create an internal replicated data grid for processing units that contain customer profile information:

```
HazelcastInstance hz = Hazelcast.newHazelcastInstance();
Map<String, CustomerProfile> profileCache =
        hz.getReplicatedMap("CustomerProfile");
```

All processing units that need access to customer-profile information should contain this code. When any processing unit updates data in the `CustomerProfile` cache, the data grid replicates the update to all other processing units that contain that same named `CustomerProfile` cache. A processing unit can contain as many in-memory replicated caches as it needs to complete its work. Alternatively, one processing unit can make a remote call to another processing unit to ask for data (choreography) or leverage the processing grid (described in the next section) to orchestrate the data request (see “Orchestration Versus Choreography” in Chapter 20 for more information about orchestration and choreography).

Using data replication within the processing units allows additional instances to start without having to read data from the database, as long as at least one instance contains the named replicated cache. When a new processing unit instance starts, it broadcasts a request through the caching provider (such as Hazelcast) to join other processing units with the same named cache. Once other processing units acknowledge the broadcast request and connect to the new processing unit, one of them (usually the first one to connect to the new processing unit) sends the cache data to the the new instance so it’s in sync with all other instances with the same named cache.

Each instance of a processing unit contains a *member list* containing the IP addresses and ports of all other processing-unit instances containing the same

named cache. For example, suppose there is a single processing-unit instance containing the customer profile functionality and its in-memory replicated cached data. In this case there is only one instance, so its member list contains only itself. This is illustrated in the following logging statements, generated using Hazelcast:

```
Instance 1:
Members {size:1, ver:1} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this
]
```

When another processing unit starts up with the same named cache, the member lists of both services are updated to reflect the IP addresses and ports of each processing unit:

```
Instance 1:
Members {size:2, ver:2} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
]

Instance 2:
Members {size:2, ver:2} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 this
]
```

When a third processing unit starts up, the member lists of instance 1 and instance 2 are both updated to reflect the new third instance:

```
Instance 1:
Members {size:3, ver:3} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 2:
Members {size:3, ver:3} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316 this
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 3:
Members {size:3, ver:3} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753 this
]
```

Now all three instances know about each other (including themselves, as denoted by the word `this` at the end of the member line). Suppose instance 1 receives a request from a customer to update their bill-to address. When instance 1 updates the cache with a `cache.put()` or similar cache update method, the data grid (such as Hazelcast) will asynchronously update the other replicated caches with the same update, ensuring that all three customer profile caches contain the new bill-to address, thus always remaining in sync with one another with the same data.

When processing-unit instances go down, all other processing-units' member lists are automatically updated to reflect the change. For example, if instance 2 goes down, the caching product immediately updates the member lists of instance 1 and 3 to remove the lost instance:

```
Instance 1:
Members {size:2, ver:4} [
```

```
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268 this
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 3:
Members {size:2, ver:4} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753 this
]
```

Replicated and distributed caching

Space-based architecture relies on caching to process transactions in applications. Space-based architecture removes the need for direct reads and writes to a database, which is how it can support high scalability, elasticity, and performance. This architecture style mostly relies on in-memory replicated caching, although it can use distributed caching as well.

With *replicated caching*, as illustrated in Figure 16-6, each processing unit contains its own in-memory data grid that is synchronized between all processing units using that same named cache. When a cache within any of the processing units is updated, the other processing units are automatically updated with the new information.

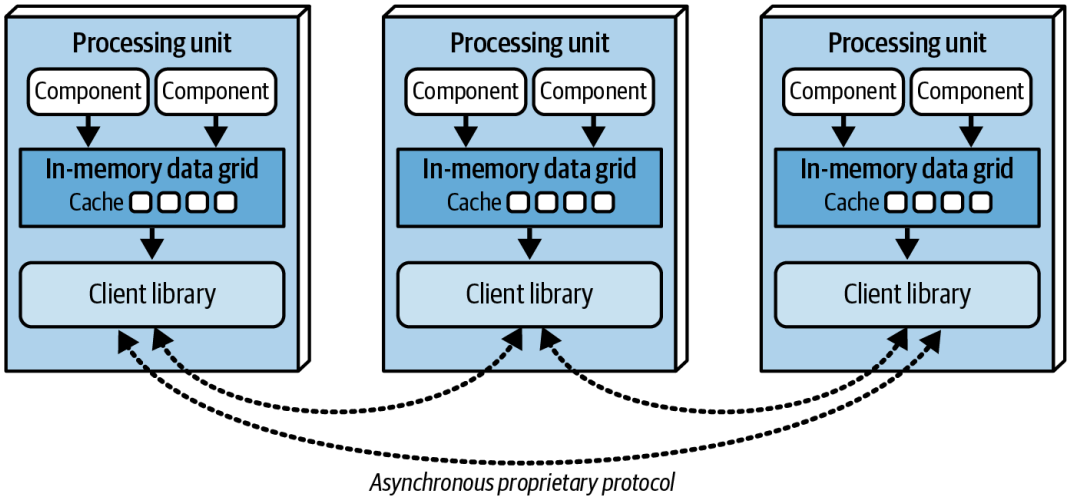

Figure 16-6. Replicated caching synchronizes in-memory caches between processing units

Not only is replicated caching extremely fast, it also supports high levels of fault tolerance. Since there is no central server holding the cache, replicated caching does not have a single point of failure.[1]

While replicated caching is the standard caching model for space-based architecture, there are some cases where it can't be used. One such situation is when the system must handle high data volumes. When internal memory caches grow larger than 100 MB, each processing unit must use so much memory that it can cause issues with elasticity and scalability. Processing units are generally deployed within a virtual machine or a container (such as Docker), each of which only has a certain amount of memory available for internal cache usage. This limits the number of processing-unit instances that can be started to process high-throughput situations.

Another situation where replicated data caches don't work well is when the cache data is updated very frequently. As shown in "Data Collisions", if the update rate of the cache data is too high, the data grid might be unable to keep up, which

could harm data consistency across all processing-unit instances. When these situations occur, most architects choose to use a distributed cache instead.

Distributed caching, as illustrated in Figure 16-7, requires an external server or service dedicated to holding a centralized cache. In this model, the processing units do not store data in internal memory. Instead, they use a proprietary protocol to access the data from the central cache server. Distributed caching supports high levels of data consistency, because the data is all kept in one place and does not need to be replicated. However, this model doesn't perform as well as replicated caching because the cache data must be accessed remotely, adding to the overall latency of the system.

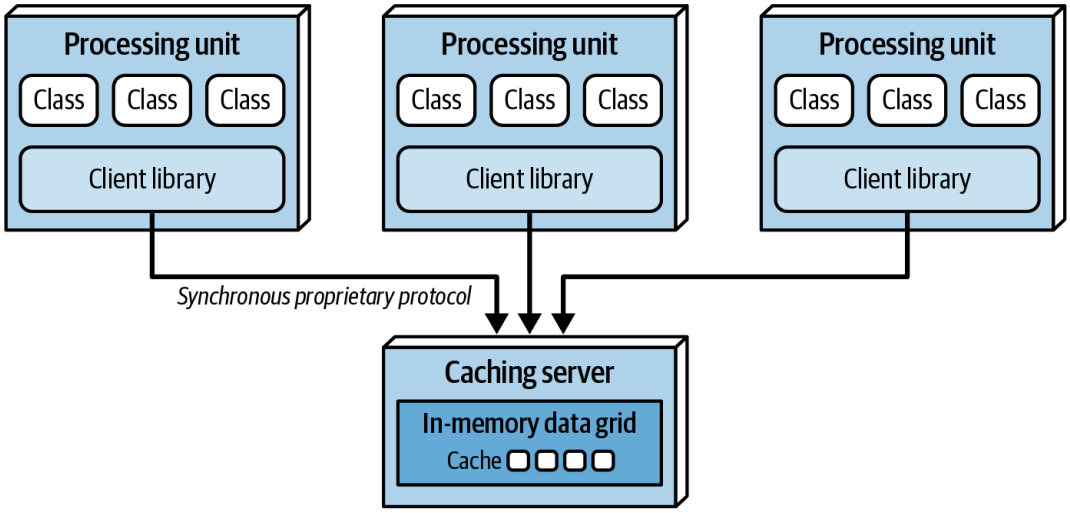

Figure 16-7. Distributed caching creates good data consistency between processing units

Fault tolerance is also an issue with distributed caching. If the cache server containing the data goes down, none of the processing units can access or update data. This single point of failure can be mitigated by mirroring the distributed cache, but this could present consistency issues if the primary cache server goes down unexpectedly and the data does not make it to the mirrored cache server.

When the size of the cache is relatively small (under 100 MB) and the update rate of the cache is low enough that the caching product's replication engine can keep up, the decision between using a replicated cache and a distributed cache becomes a question of prioritizing data consistency versus performance and fault tolerance. A distributed cache will always offer better data consistency over a replicated cache, because the data is in one place, not spread across multiple processing units. However, performance and fault tolerance will always be better with a replicated cache. Many times, the deciding factor ends up being the *type* of data being cached in the processing units. If the system's primary need is for highly consistent data (such as inventory counts of the available products), that usually warrants a distributed cache. If data does not change often (such as reference data like name/value pairs, product codes, and product descriptions), consider using a replicated cache for quick lookup. Table 16-1 summarizes some criteria for choosing when to use a distributed versus a replicated cache.

Table 16-1. Distributed versus replicated caching

Decision criteria	Replicated cache	Distributed cache
Optimization	Performance	Consistency
Cache size	Small (<100 MB)	Large (>500 MB)
Type of data	Relatively static	Highly dynamic
Update frequency	Relatively low	High update rate

Decision criteria	Replicated cache	Distributed cache
Fault tolerance	High	Low

When choosing which caching model to use with space-based architecture, remember that in most cases, *both* models will be applicable. In other words, neither replicated caching nor distributed caching solves every problem. Different processing units can also use different models. Rather than compromising by choosing a single consistent caching model across the application, leverage each model for its strengths. For example, for a processing unit that maintains the current inventory, choose a distributed caching model for data consistency; for a processing unit that maintains the customer profile, choose a replicated cache for performance and fault tolerance.

Near-cache considerations

A *near-cache* is a hybrid caching model bridging in-memory data grids with a distributed cache. In this model (illustrated in Figure 16-8) the distributed cache is referred to as the *full backing cache*, and each in-memory data grid contained within a processing unit is called a *front cache*. The front cache always contains a smaller subset of the full backing cache, and uses an *eviction policy* to remove older items so that newer ones can be added. The front cache has three eviction policy options: a *most recently used* cache, containing the most recently used items; a *most frequently used* cache, containing the most frequently used items; or a *random replacement* eviction policy, which removes items randomly when space is needed. Random replacement is a good eviction policy when there is no clear analysis of the data providing a reason to keep either the latest used or the most frequently used.

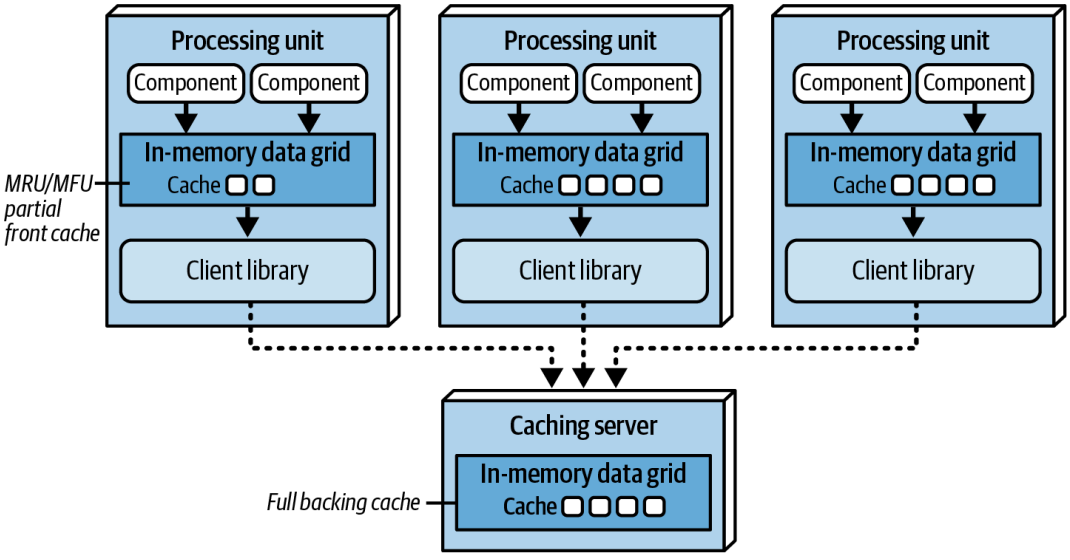

Figure 16-8. The near-cache model uses both a front cache and a backing cache

While the front caches are always kept in sync with the full backing cache, the front caches contained within each processing unit are not synchronized between other processing units that share the same data. This means that multiple processing units that share the same data context (such as a customer profile) will likely all have different data in their front caches. This creates inconsistencies in performance and responsiveness between processing units, so we do not recommend using a near-cache model in a space-based architecture.

Processing Grid

The *processing grid*, illustrated in Figure 16-9, is an optional component within the virtualized middleware that manages orchestrated request processing when multiple processing units are involved in a single business request. If a request requires coordination between more than one type of processing unit (such as an order-processing unit and a payment-processing unit), the processing grid mediates and orchestrates the request between those two processing units.

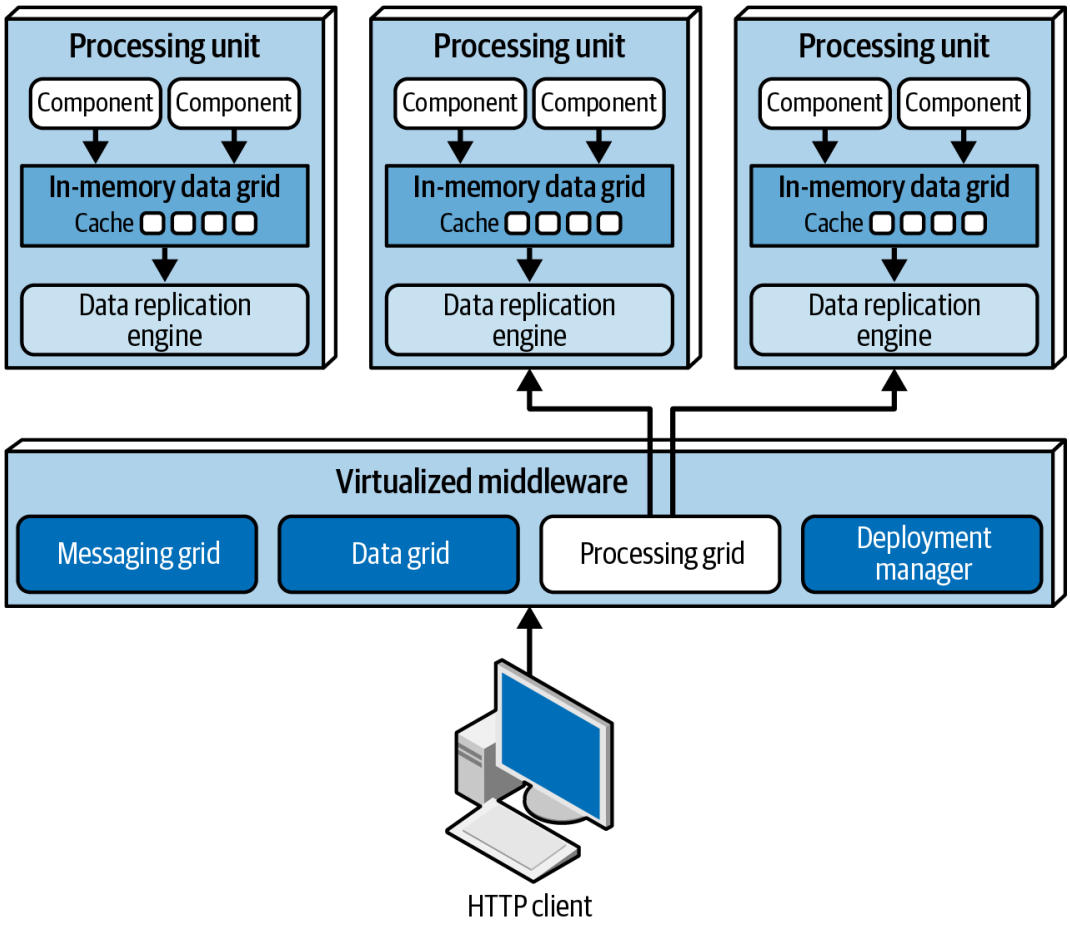

Figure 16-9. The processing grid manages orchestration between processing units

Most modern space-based implementations (particularly those with fine-grained services) implement their processing-grid functionality through separate, fine-grained *orchestration processing units* rather than a single coarse-grained orchestration engine, with each orchestration processing unit handling a single major workflow. For an ecommerce example, let's say that when a customer places an order, three processing units must coordinate: *Order Placement*, *Payment*, and *Inventory Adjustment*. The architect could create an *Order Placement Orchestrator* processing unit to orchestrate these three processing units. They might also create separate orchestration processing units for other major workflows, such as handling order returns and replenishing stock.

Deployment Manager

The `Deployment Manager` component manages the dynamic startup and shutdown of processing-unit instances based on load conditions. This component continually monitors response times and user loads, starts up new processing units when load increases, and shuts down processing units when the load decreases. It is critical to achieving variable scalability (elasticity) within an application. Most cloud-

based infrastructures handle this responsibility, as do service-orchestration products such as Kubernetes.

Data Pumps

A *data pump* is a way of sending data to another processor, which then updates a database. Space-based architectures need data pumps because processing units do not read from and write to databases directly. Data pumps in space-based architecture are always asynchronous, providing eventual consistency between the in-memory cache and the database. When a processing-unit instance receives a request and updates its cache, that processing unit becomes the owner of the update, and therefore responsible for sending it through the data pump so that the database can be updated eventually.

Data pumps are usually implemented in a space-based architecture using messaging, as shown in Figure 16-10.

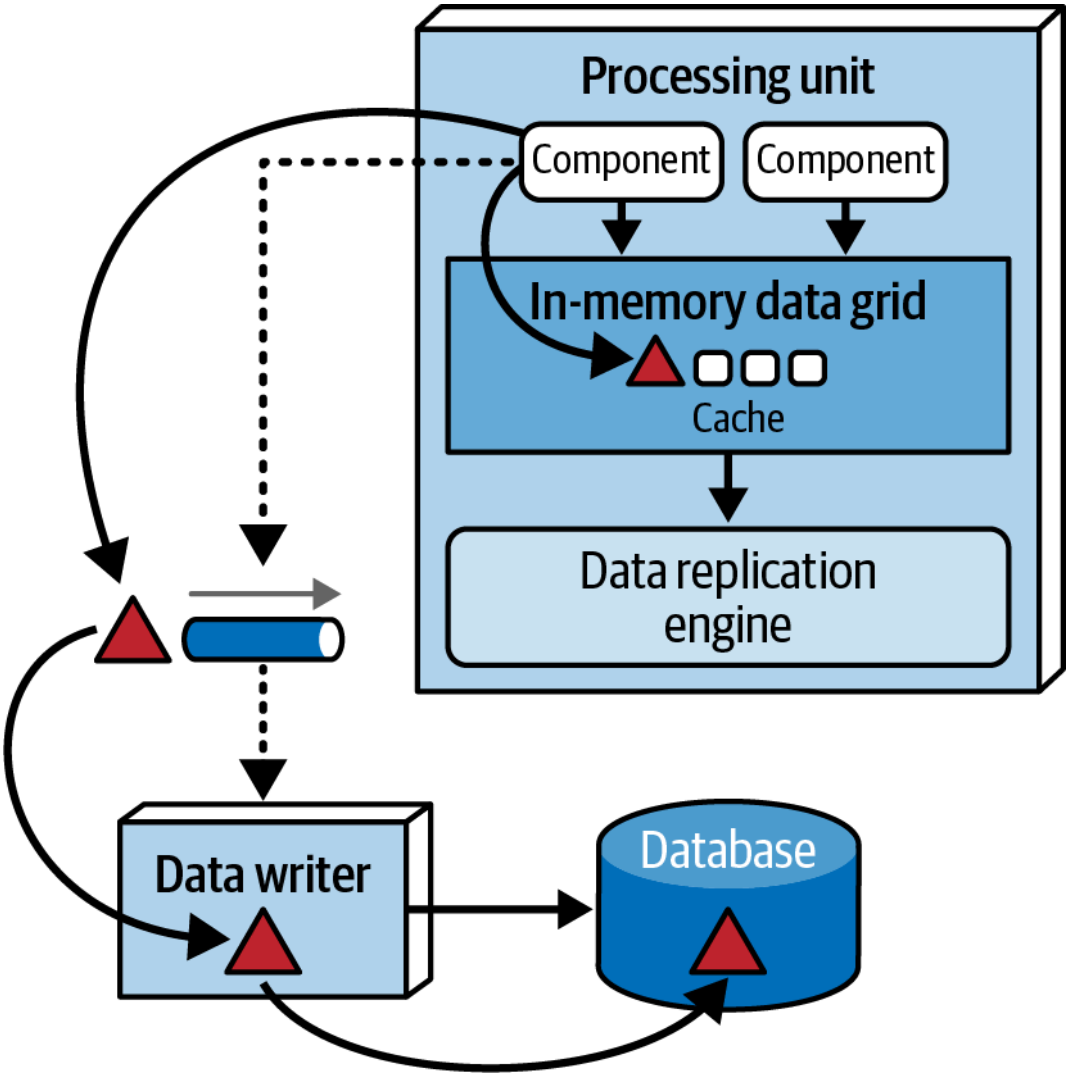

Figure 16-10. Data pumps are used to send data to a database

Messaging supports not only asynchronous communication, but also guaranteed delivery, message persistence, and message order, through first-in, first-out (FIFO) queuing. Furthermore, messaging decouples the processing unit and the data writer so that if the data writer is unavailable, processing within the processing units isn't interrupted.

Most space-based architectures have multiple data pumps. Usually each one is dedicated to a particular domain or subdomain (such as customer or inventory), but they can be dedicated to each type of cache (such as `CustomerProfile`, `CustomerWishlist`, and so on) or to a processing-unit domain (such as `Customer`) that also contains a much larger general cache.

Data pumps usually have contracts, including an action associated with the contract data (add, delete, or update). The contract can be a JSON schema, XML schema, an object, or even a *value-driven message* (a map message containing name-value pairs). For updates, the data pump's message payload usually only contains the new data values. For example, if a customer changed a phone number on their profile, only the new phone number would be sent, along with the customer ID and an action to update the data.

Data Writers

The `Data Writer` component accepts messages from a data pump and updates the database with the information in their payloads (see Figure 16-10). Data writers can be implemented as services, applications, or data hubs (such as Ab Initio). Their granularity can vary, based on the scope of the data pumps and processing units.

A *domain-based data writer* contains the necessary database logic to handle all updates within a particular domain (such as order processing), regardless of the number of data pumps it is reading from. The system in Figure 16-11 has four different processing units and four different data pumps to represent the customer domains (`Profile`, `WishList`, `Wallet`, and `Preferences`)—but only one data writer. That single customer data writer listens to all four data pumps and contains the database logic (such as SQL) to update customer-related data in the database.

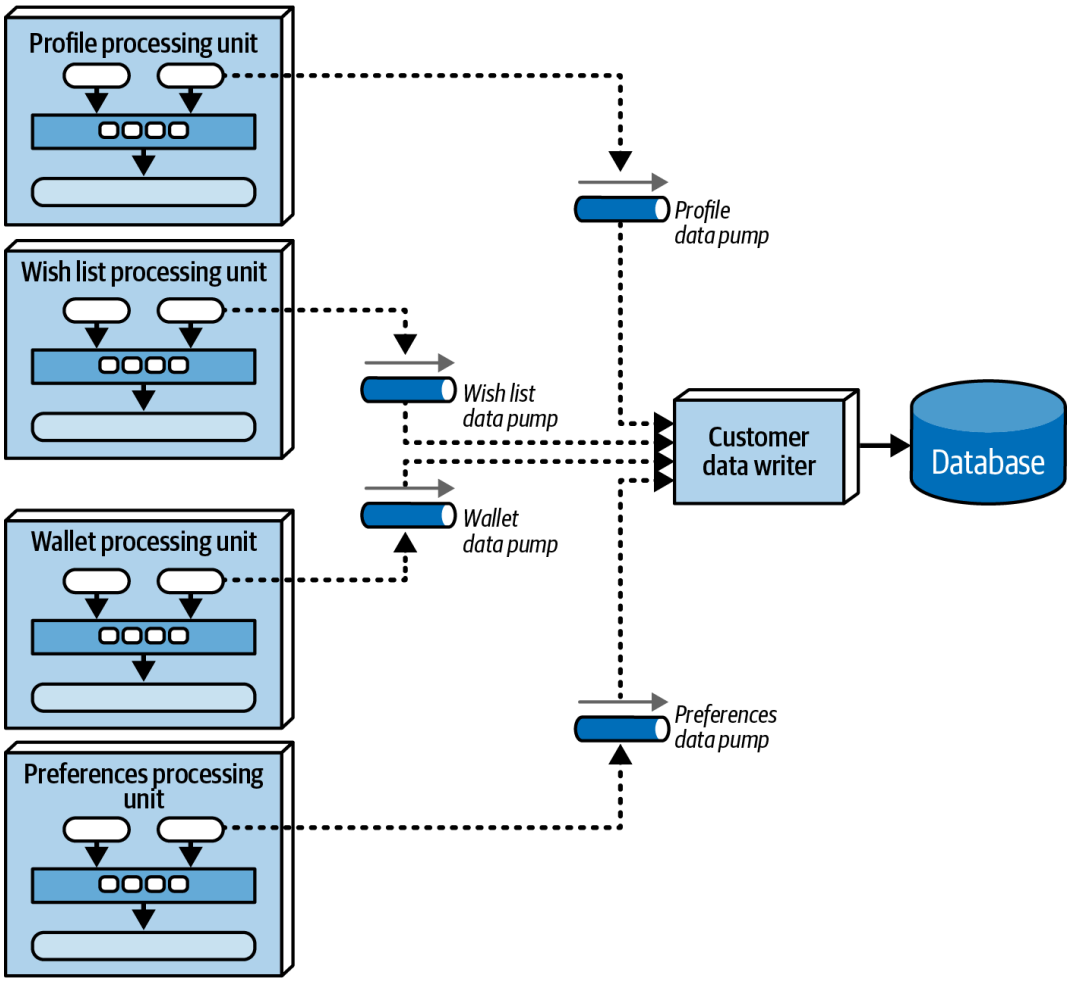

Figure 16-11. Domain-based data writer

Alternatively, each class of processing unit can have its own dedicated `Data Writer` component, as illustrated in Figure 16-12. In this model, each data writer is dedicated to a corresponding data pump and contains only the database processing logic for that particular processing unit (such as `Wallet`). While this model tends to produce a lot of data-writer components, it does provide better scalability and agility because it aligns the processing unit, data pump, and data writer.

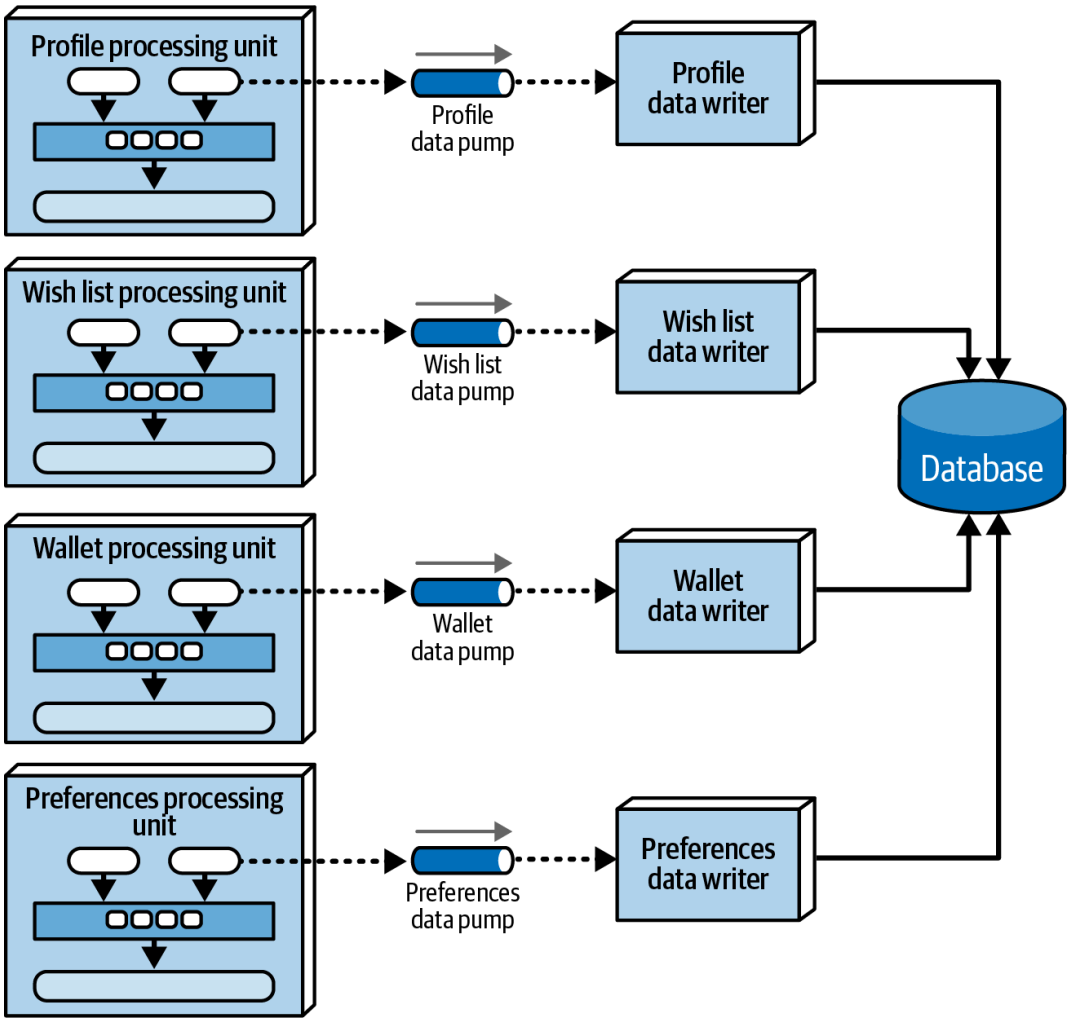

Figure 16-12. Dedicated data writers for each data pump

Data Readers

While data writers are responsible for updating the database, *data readers* read data from the database and send it to the processing units via a *reverse data pump* (which we'll discuss more in a moment). In space-based architectures, data readers are only invoked in one of three situations: all processing unit instances of the same named cache crash, all processing units within the same named cache are redeployed, or archive data not contained in the replicated cache must be retrieved.

If all instances come down due to a system-wide crash or redeployment, data must be loaded in the cache from the database—something we generally try to avoid in space-based architecture. When instances of a class of processing unit start coming back up, each one will try to grab a lock on the cache. The first one to get the lock becomes the temporary cache owner; the others go into a wait state until the lock is released. (This might vary based on the type of cache implementation, but regardless, there is one primary owner of the cache in this scenario.) To load the cache, the temporary cache owner sends a message to a queue, requesting data. The `Data Reader` component accepts the read request and then performs the necessary database-query logic to retrieve the needed data. It sends that data to a different queue called a reverse data pump, which sends it to the temporary cache owner processing unit. Once it loads the cache, the temporary owner releases the lock. All other instances are then synchronized, and processing can begin. This processing flow is illustrated in Figure 16-13.

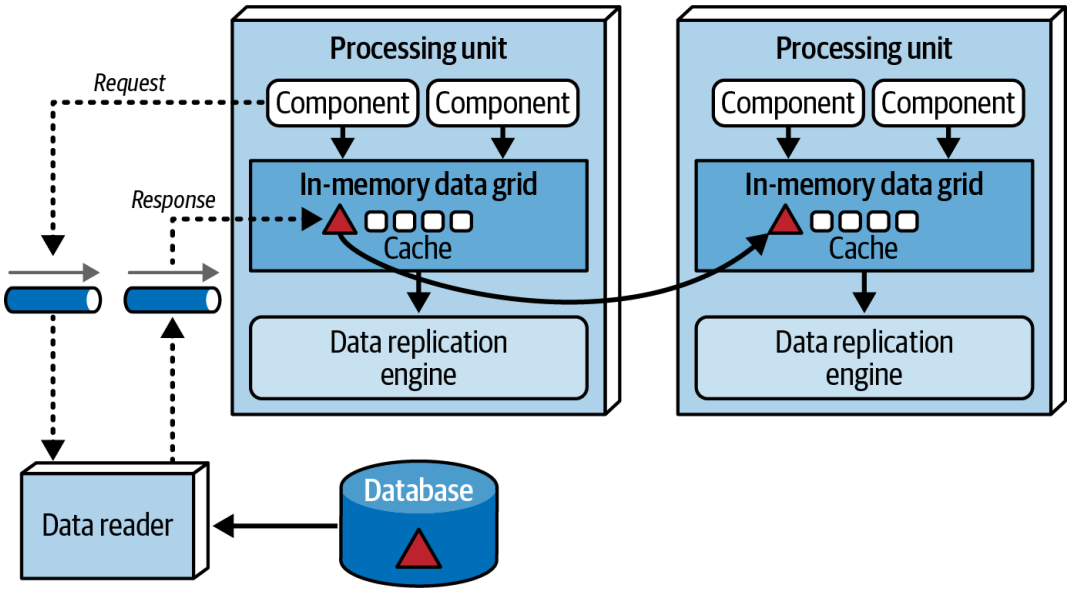

Figure 16-13. Data readers send data to the processing units

Like data writers, data readers can be domain based or dedicated to a specific class of processing unit (the latter is more common). Its implementation is also the same as the data writers—service, application, or data hub.

The data writers and data readers essentially form a *data abstraction layer* (or *data access layer*, in some cases). The difference between the two is in the amount of detailed knowledge the processing units have with regard to the database schema (the structure of the tables). With a data access layer, the processing units are coupled to the underlying data structures in the database, and only access the database indirectly, through the data readers and writers. With a data abstraction layer, on the other hand, the processing unit is decoupled from the underlying schemas through the use of separate contracts.

Space-based architecture generally relies on a data abstraction layer model so that the replicated cache schema in each processing unit can be different from the underlying database schemas. This means that incremental changes to the database don't necessarily affect the processing units. To facilitate this incremental change, the data writers and readers contain transformation logic. If a column type changes or a column or table is dropped, the data readers and writers can buffer the database change until the necessary changes can be made to the processing-unit caches.

Data Topologies

Since processing units don't interact with the database directly, space-based architecture is tremendously flexible in the database topologies it can use. The use of asynchronous data pumps combined with data readers and writers means request processing (transactional) is largely independent of the database, giving the architect a wide variety of choices regarding the database topology and database type.

The choice of database topology is influenced by a host of factors. In a space-based architecture, the primary deciding factor is how the system will use the backing database. For example, if reporting and data analytics are especially important, a monolithic database topology might be more effective—unless the reporting and data analytics are done through a data mesh, in which case a domain-based database topology might be better.

Throughput and overall domain-based data consistency are also considerations when choosing a database topology. A single monolithic database might prove to be a bottleneck during synchronization, slowing overall synchronization time and detracting from data consistency; a domain-based database topology might offer better overall synchronization time and hence data consistency, if the data can be cleanly domain partitioned. Finally, if downstream systems need to use the database for further processing, a monolithic database topology might be a better fit.

Cloud Considerations

Space-based architecture offers some unique options when it comes to deployment environments. The entire system—including the processing units, virtualized middleware, data pumps, data readers and writers, and the database—can be deployed within cloud-based environments or on-premises (on-prem). However, this architecture style can also be deployed in *both of these environments at once* (as illustrated in Figure 16-14), a unique, powerful feature not found in other architecture styles. In this hybrid topology, applications are deployed via processing units and virtualized middleware in managed cloud-based environments, while the physical databases and corresponding data are kept on-prem. This supports very effective cloud-based data synchronization, thanks to the asynchronous data pumps and eventual consistency model of this architecture style. Transactional processing can take place in dynamic, elastic cloud-based environments, while physical data management, reporting, and data analytics stay in secure on-prem environments.

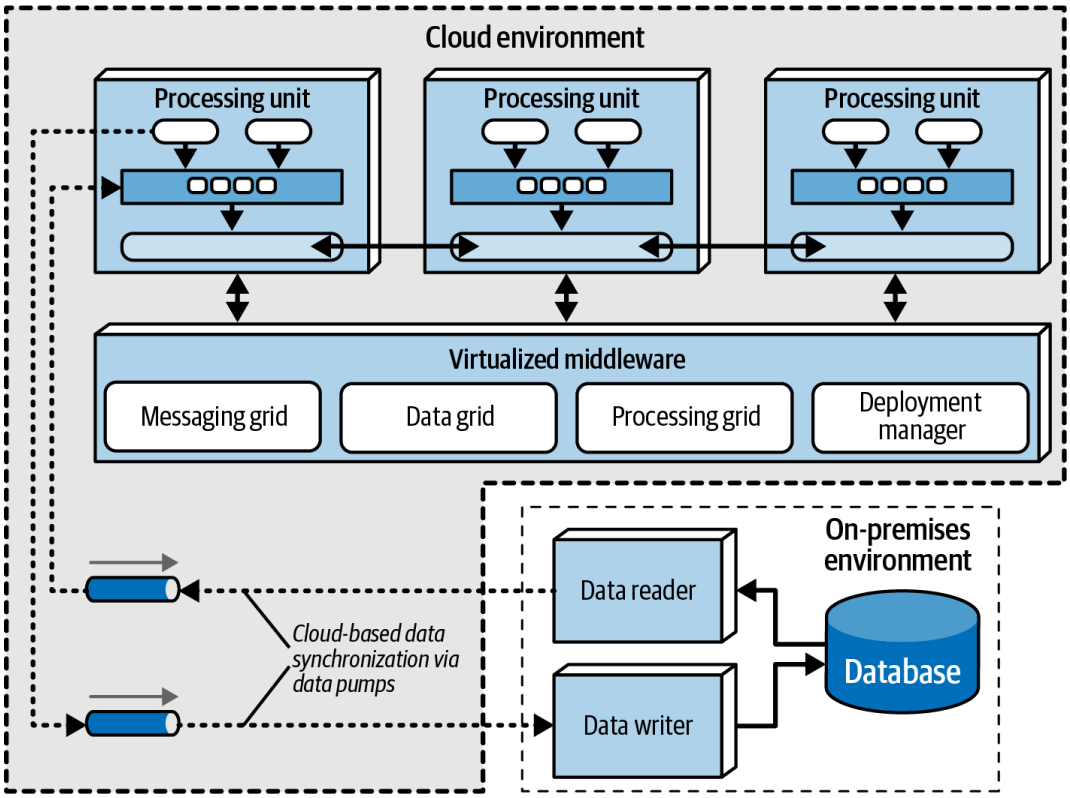

Figure 16-14. Hybrid cloud-based and on-prem topology

The elastic nature of cloud infrastructure and cloud-based services—hybrid or not—matches the shape of this architectural style well, making cloud-based environments a good choice for space-based architectures.

Common Risks

Not surprisingly, most of the risks associated with the space-based architectural style have to do with data, given its use of caching and background data synchronization. The following sections describe some common risks.

Frequent Reads from the Database

Space-based architecture achieves high scalability and concurrency by using caching for all transactional data. This prevents the system from doing excessive database reads and writes, which can lower the system's overall scalability, elasticity, and responsiveness, making it difficult to realize the benefits of this architectural style.

Database reads typically only occur in two scenarios: reading archived data (such as order history or past bank statements), or *cold-starting* a processing unit (the initial start of a processing unit when no other instances are running). If the cached data volumes are so high that most data needs to be archived and retrieved from the backing database, or if processing units crash or are redeployed frequently, this might not be the right architecture for the problem domain.

Data Synchronization and Consistency

Because data pumps and data writers synchronize data between the in-memory cache and the database, data will always be eventually consistent in a space-based architecture. However, because this style is typically used in situations with a very high concurrent user load, bottlenecks in the data pump are common. These bottlenecks can significantly delay data from making it to the backing database. This can be a significant risk if downstream systems need to have updated data quickly.

Another inherent risk related to data synchronization is data loss within the data pump. This risk is usually mitigated by using *persisted queues* (in which data in a queue is stored on disk as well as in memory) and using client-acknowledgment mode in the data writers when reading from the data pump. Client-acknowledgment mode keeps the message in the queue until the data writer acknowledges to the message broker that it has completed processing. During message processing, the message broker ensures that no other data writer can read the in-process message. While these techniques help prevent data loss in the data pump, they can also slow down overall responsiveness and degrade data consistency.

High Data Volumes

Because all transactional memory is cached in the processing units, data volumes need to remain relatively low, particularly as more instances of a processing unit are added. This makes it critical to pay close attention to the size of the in-memory cache to avoid a processing unit running out of memory and hence crashing.

Data Collisions

A *data collision* occurs when data is updated in one cache instance (cache A), and during replication to another cache instance (cache B), the same data is updated by that cache (cache B). This can happen when using replicated caching in an

active/active state, meaning that multiple processing units can update the same data at the same time. Collisions typically occur when the update rate of the data in the cache exceeds the *replication latency (RL)*—the time it takes to synchronize each same-named cache. In this scenario, the local update to cache B will be overridden by the old data from cache A, and the same data in cache A will be overridden by the update from cache B. This makes the data in each cache inconsistent.

To demonstrate the data collision problem, let's assume there are two service instances (A and B) of an order placement service, each containing a replicated cache of product inventory (blue widgets). The flow is as follows:

1. The current inventory count is 500 units in each instance A and B.
2. Instance A receives a request from a customer to purchase 10 units, and updates the inventory cache for blue widgets to 490 units.
3. Before instance A's data can be replicated to instance B, instance B receives a purchase request for 5 units, and updates the inventory cache for blue widgets to 495 units.
4. The cache in instance B gets updated to 490 units due to replication from instance A's update.
5. The cache in instance A gets updated to 495 units due to replication from instance B's update.
6. Both caches are incorrect and out of sync: the inventory should be 485 units in each of the instances.

Several factors that influence how many data collisions might occur: the number of processing-unit instances that contain the same cache, the cache's update rate, the cache's size, and the RL of the caching product. We can probabilistically determine how many potential data collisions might occur based on these factors, using the following formula:

$$CollisionRate = N * \frac{UR^2}{S} * RL$$

$$CollisionRate = N * \frac{UR^2}{S} * RL$$

Here, *N* represents the number of service instances using the same named cache. *UR* represents the update rate in milliseconds (squared), *S* is the cache size (in number of rows), and *RL* is the caching product's replication latency (in milliseconds).

This formula is useful for determining the percentage of data collisions that will likely occur based on updates within a given time frame (for example, each hour), and thus how feasible it would be for this system to use replicated caching. For example, consider the values shown in Table 16-2 for the factors involved in this calculation.

Table 16-2. Base values

Update rate (UR):	20 updates/second
Number of instances (N):	5
Cache size (S):	50,000 rows
Replication latency (RL):	100 milliseconds
Updates:	72,000 per hour

Collision rate:	14.4 per hour
Percentage:	0.02%

Applying these factors to the formula yields 72,000 updates an hour, with a high probability that 14 updates to the same data *may* collide. Given the low percentage (0.02%), replication would be a viable option.

Variations in RL can have a significant impact on data consistency. Replication latency depends on many factors, including the type of network and the physical distance between processing units. RL values must be calculated and derived from actual measurements in a production environment, which is why they're rarely published. The value of 100 ms used in the prior example is a good planning number if the actual RL is unavailable. For example, changing the RL from 100 ms to 1 ms yields the same number of updates (72,000 per hour) but produces a much lower probability of collisions occurring (0.1 collisions per hour). This scenario is shown in Table 16-3.

Table 16-3. Impact on replication latency

Update rate (UR):	20 updates/second
Number of instances (N):	5
Cache size (S):	50,000 rows
Replication latency (RL):	1 millisecond (changed from 100)
Updates:	72,000 per hour
Collision rate:	0.1 per hour
Percentage:	0.0002%

The number of processing units that contain the same named cache (as represented by *N*) also has a direct proportional relationship to the number of possible data collisions. For example, reducing the number of processing units from 5 instances to 2 instances yields a data-collision rate of only 6 per hour, out of 72,000 updates per hour, as shown in Table 16-4.

Table 16-4. Impact on the number of processing-unit instances

Update rate (UR):	20 updates/second
Number of instances (N):	2 (changed from 5)
Cache size (S):	50,000 rows
Replication latency (RL):	100 milliseconds
Updates:	72,000 per hour
Collision rate:	5.8 per hour
Percentage:	0.008%

The cache size is the only factor that is inversely proportional to the collision rate: as the cache size decreases, collision rates increase. In this example, reducing the cache size from 50,000 rows to 10,000 rows (and keeping everything else the same as in the first example) yields a collision rate of 72 per hour, significantly higher than with 50,000 rows, as shown in Table 16-5.

Table 16-5. Impact on cache size

Update rate (UR):	20 updates/second
Number of instances (N):	5
Cache size (S):	10,000 rows (changed from 50,000)
Replication latency (RL):	100 milliseconds

Updates:	72,000 per hour
Collision rate:	72.0 per hour
Percentage:	0.1%

Under normal circumstances, most systems do not have consistent update rates over a long period of time (such as an eight-hour day). When using this calculation, we recommend understanding your system's maximum update rate during peak usage and calculating minimum, normal, and peak collision rates.

Governance

Space-based architecture, with its many moving parts, is a complicated architectural style to design and implement. Proper governance is critical to ensuring success—especially for controlling memory consumption, given this style's issues with internal memory usage (as we noted earlier, in "Common Risks").

To address these memory issues, we recommend writing a continuous, automated governance fitness function to have each instance of a processing unit make its current memory usage observable periodically. Since all instances of a processing unit have the same replicated cache, the fitness function only needs to report on the name of the processing unit. Use a separate fitness function to record the number of instances for each processing unit, allowing you to calculate the total memory consumption per processing unit. Figure 16-15 shows an example of this continuous fitness function's output.

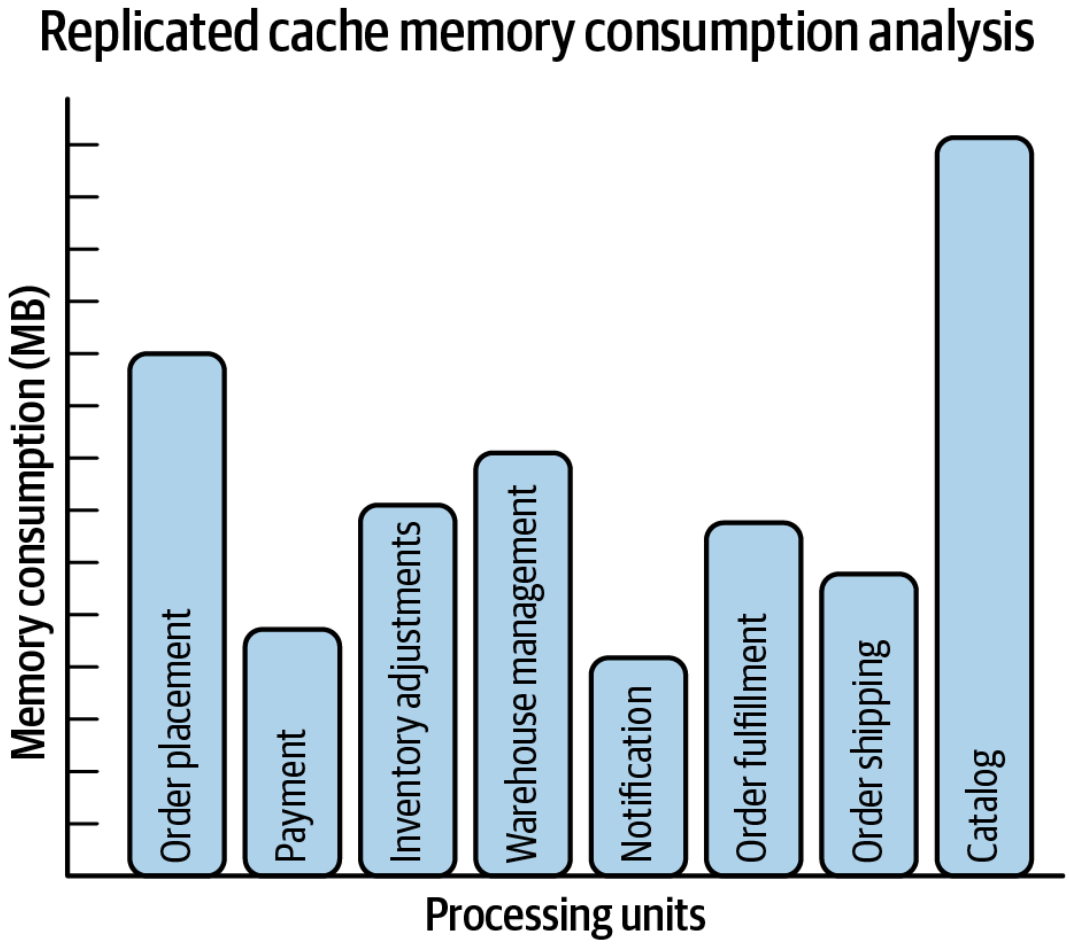

Figure 16-15. An example fitness function to track memory consumption

Another useful governance strategy to maintain overall data consistency is to track and measure synchronization time—specifically, how long it takes a cache update to sync to the corresponding database. A good method is to have each processing unit stream the request ID of an update along with its corresponding timestamp, and have each data writer stream the same request ID along with the timestamp after the database commit. Then, write a fitness function to associate these request IDs and subtract the timestamps to calculate the synchronization time. The fitness function could track this at an atomic level, using times associated with a particular processing unit, or holistically, by averaging the overall synchronization times. Analyzing these trends helps architects track the effects of changes to the architecture, such as whether the changes are making synchronization times better or worse, and whether the architecture is meeting the business's synchronization timing goals. Figure 16-16 illustrates this kind of governance through a continuous fitness function.

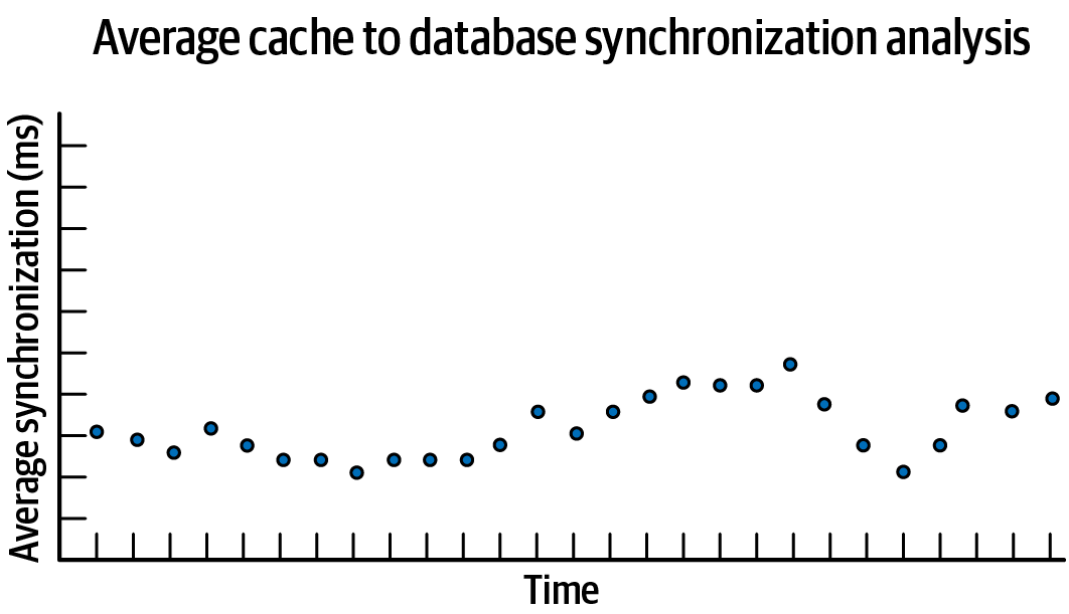

Figure 16-16. An example fitness function to track aggregated average synchronization times

As we've noted, because data pumps act as a backpressure point in space-based architecture and because database writes take longer than cache writes, data pumps can form a bottleneck in the overall system. To govern this, we recommend tracking and measuring this kind of bottleneck when it arises. First, to determine the extent of the bottleneck, write a fitness function to track the queue depth of the queue used in the data pumps. Too much of a bottleneck increases the synchronization time (and thus data consistency) and also increases the chances of data loss and data collisions, particularly during periods of high user-concurrency levels. Like the previous fitness function, this function can report atomically about each data-pump queue or aggregate the overall system average. Figure 16-17 shows a bottleneck analysis for an *Order Placement* processing unit and its corresponding data pump in a typical order-processing system.

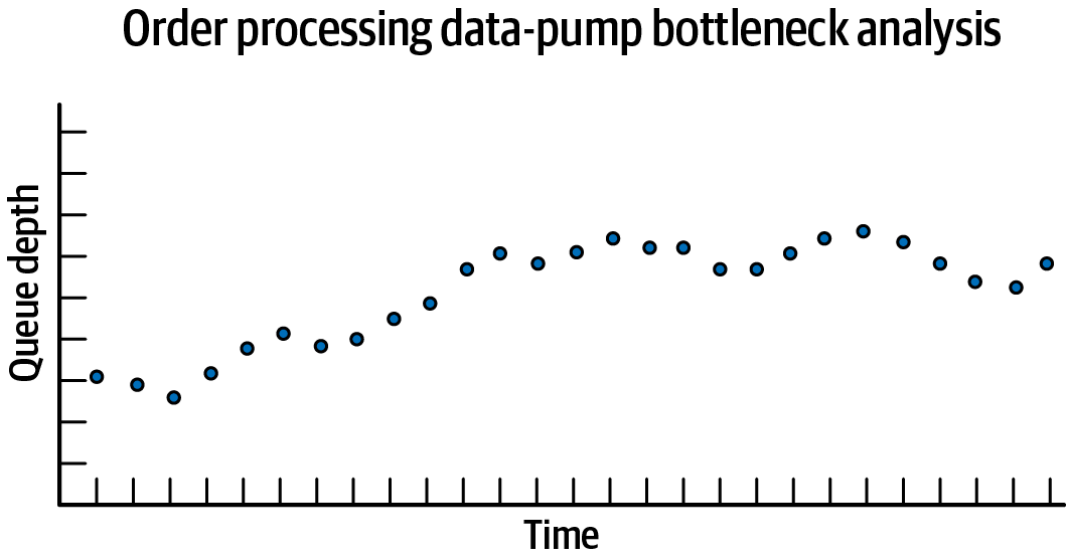

Figure 16-17. An example fitness function to track data-pump bottlenecks

Other governance fitness functions for space-based architecture could track the frequency of reads to the database (requests to data readers), which can affect scalability, elasticity, and responsiveness. It's also a good idea to use fitness functions to measure scalability, elasticity, and responsiveness: since these architectural characteristics are the main reasons to use a space-based architecture, it makes sense to track and measure them.

Team Topology Considerations

Although space-based architecture is largely considered a technically partitioned architecture due to the many artifacts that make up any particular domain or subdomain, it's most effective with technically partitioned teams that are aligned with its technical areas (such as functionality, data pumps, data readers and writers, and backend database management). However, it can still work well when teams are aligned by domain area (such as cross-functional teams with specialization). Here are some things to consider about aligning the specific team topologies outlined in "Team Topologies and Architecture" with space-based architecture:

Stream-aligned teams

> Depending on the size of the system, stream-aligned teams might find themselves struggling to implement domain-based changes based on the technical partitioning in this architectural style. For example, a stream-based change might impact one or more processing units, data pumps, data readers, data writers, cache contracts, or orchestrators, as well as the backing database. This can be a lot for a single stream-based team to manage, particularly if those artifacts are shared by other teams. The larger and more complex the system, the less effective stream-aligned teams will be with space-based architecture.

Enabling teams

> Because some of this style's artifacts (data pumps, data readers, data writers, and virtualized middleware) may be shared or cross-cutting, it's a good fit for enabling teams. Dedicating a team to a particular artifact (such as a data writer and its corresponding data pump) can allow its members to experiment and find ways of making these artifacts more efficient,

independent of the team working on the primary functionality in a particular processing unit.

Complicated-subsystem teams

Complicated-system teams can leverage the space-based architecture style's technically partitioned nature to focus on one part of the system (such as the data grid or data pumps). Some of these artifacts can get extremely complex, so they lend themselves well to the complicated-subsystem team topology. Dealing with data collisions (see "Data Collisions") and other sorts of asynchronous data synchronization errors in the data writers is quite complex; this is a great example of a complicated subsystem that functional domain-based teams working on processing units shouldn't need to concern themselves about.

Platform teams

As with most architectural styles, developers working on a space-based architecture can leverage the benefits of the platform-team topology by utilizing common tools, services, APIs, and tasks, especially if the infrastructure-related parts of the architecture (such as the data pumps and virtualized middleware) are considered platform related.

Style Characteristics

A one-star rating in the characteristics ratings table in Figure 16-18 means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. Definitions for each characteristic identified in the scorecard can be found in Chapter 4.

	Architectural characteristic	Star rating
	Overall cost	$$$$
Structural	Partitioning type	Technical
	Number of quanta	1 to many
	Simplicity	★
	Modularity	★★★
Engineering	Maintainability	★★★
	Testability	★
	Deployability	★★★
	Evolvability	★★★
Operational	Responsiveness	★★★★★
	Scalability	★★★★★
	Elasticity	★★★★★
	Fault tolerance	★★

Figure 16-18. Space-based architecture characteristics ratings

Space-based architecture maximizes elasticity, scalability, and performance—hence we've given all of these characteristics five-star ratings. These are the driving characteristics and main advantages of this architecture style. Leveraging in-memory data caching and removing the database as a constraint can give systems built with this architecture style the high levels of elasticity, scalability, and performance they need to process millions of concurrent users.

The trade-offs for these advantages involve the system's overall simplicity and testability. Space-based architectures are very complicated, due to their use of caching and eventual consistency of the primary data store, which is the ultimate system of record. Given this style's numerous moving parts, architects should take special care to avoid data loss in the event of a crash (see "Preventing Data Loss" in Chapter 15).

Testability gets a one-star rating due to the complexity of simulating the high levels of scalability and elasticity this style supports. Testing hundreds of thousands of concurrent users at peak load is a very complicated and expensive task, so most high-volume testing occurs within production environments, with actual extreme load, incurring significant risk for normal operations.

Cost is another trade-off. Space-based architecture is relatively expensive, mostly due to its overall complexity, licensing fees for caching products, and the resource

utilization needed within cloud and on-prem systems to support its high scalability and elasticity.

While processing units, being separately deployed, might lend themselves to a level of domain partitioning, we have identified space-based as a technically partitioned architecture, since any given domain is represented by different technical components, such as processing units, data pumps, data readers and writers, and the database.

The number of quanta within a space-based architecture can vary, based on how the UI is designed and how processing units communicate. Because the processing units do not communicate synchronously with the database, the database itself is not part of the quantum equation. As a result, quanta within a space-based architecture are typically delineated through the associations between the various UIs and the processing units. Processing units that synchronously communicate (with each other or through the processing grid for orchestration) would all be part of the same architectural quantum.

Examples and Use Cases

Space-based architecture is well suited for applications that experience high spikes in user or request volume and applications that have throughput in excess of 10,000 concurrent users. We'll look at two space-based architecture use cases here: an online concert-ticketing application and an online auction system. Both require high levels of performance, scalability, and elasticity.

Concert Ticketing System

Concert ticketing systems are a unique problem domain, in that concurrent user volume is relatively low until a popular concert is announced. Once tickets for an especially popular entertainer go on sale, user volumes usually spike from several hundred concurrent users to several thousand or even tens of thousands, depending on the concert, all trying to acquire good seats! Tickets usually sell out in a matter of minutes, so these systems really require the characteristics supported by space-based architecture.

There are many challenges associated with this sort of system. First, only a certain number of tickets are available in total, regardless of the seating preferences. Seating availability must be updated continually and as fast as possible, given the high number of concurrent requests. Continually accessing a central database synchronously would be unlikely to work—it would be very difficult for a typical database to handle tens of thousands of concurrent requests through standard transactions at this scale and with this update frequency.

Space-based architecture would be a good fit for a concert ticketing system, due to its high elasticity requirements. The *deployment manager* would immediately recognize a sudden increase in the number of concurrent users and start up lots of processing units to handle the large volume of ticket purchase requests. Optimally, the deployment manager could be configured to start the necessary number of processing units shortly *before* tickets go on sale, so they would be on standby right before the significant increase in user load.

Online Auction System

Online auction systems (sites for bidding on items within an auction, such as eBay) share many characteristics with the online concert-ticketing systems we just described. Both require high levels of performance and elasticity, and both have unpredictable spikes in user and request load. When an auction starts, there is no way to determine how many people join or how many concurrent bids will occur for each asking price.

Space-based architecture is well suited for this problem domain because it allows for multiple processing units to be started as the load increases, then destroyed as the auction winds down and they're no longer needed. Individual processing units can be devoted to each auction, ensuring consistency in the bidding data. Also, the asynchronous nature of the data pumps means that bidding data can be sent to other processing (such as bid history, bid analytics, and auditing) without much latency, increasing the overall performance of the bidding process.

Space-based architecture is a complicated but very powerful architectural style. It is the only architectural style that maximizes the combination of responsiveness, scalability, and elasticity, primarily because of how it uses caching and its lack of direct database access. As such, it can be considered a specialized architectural style, used in situations that must maximize these particular architectural characteristics.

[1] There may be exceptions to this rule based on the implementation of the caching product used. Some caching products require an external controller to monitor and control data replication between processing units, but most are moving away from this model.