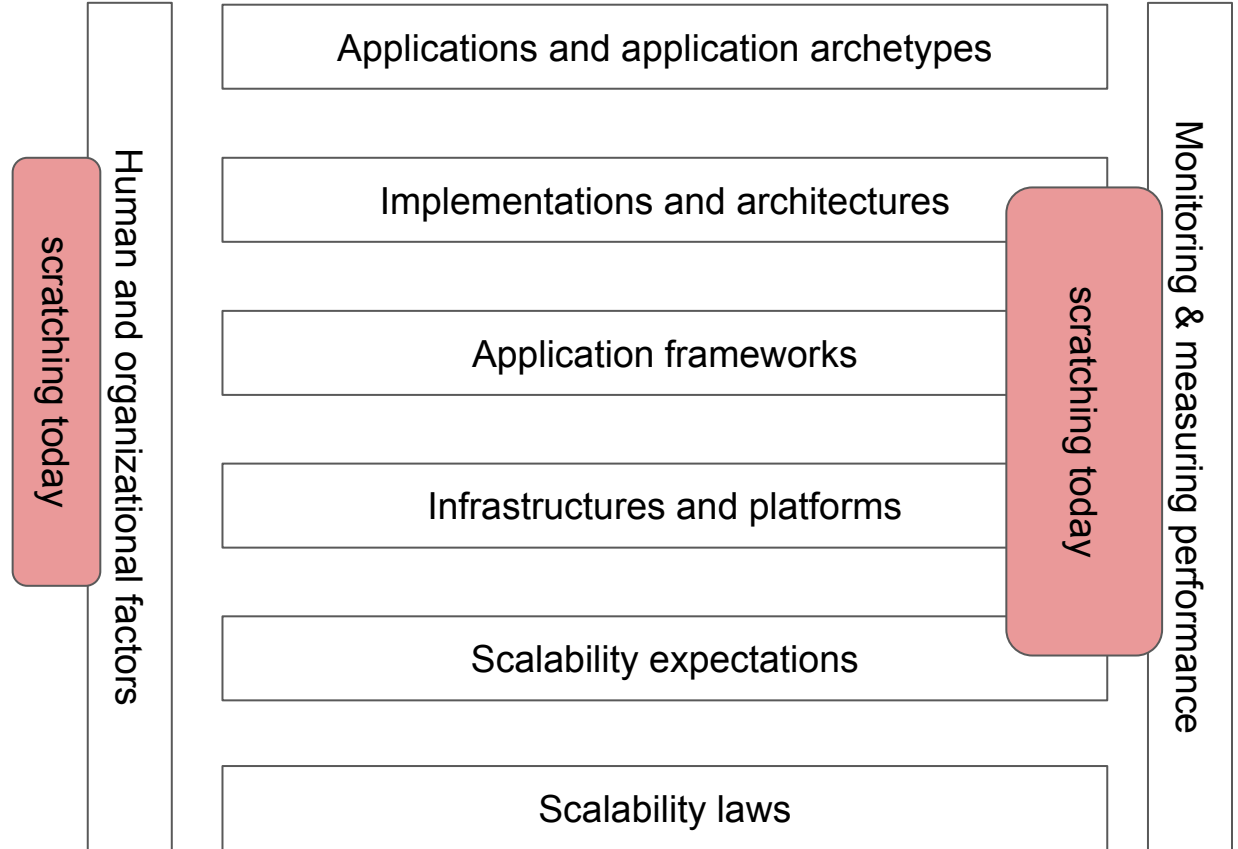


Designing and Building Scalable Web Applications

Lecture 5 / 21.11.2022

The Big Picture



Agenda

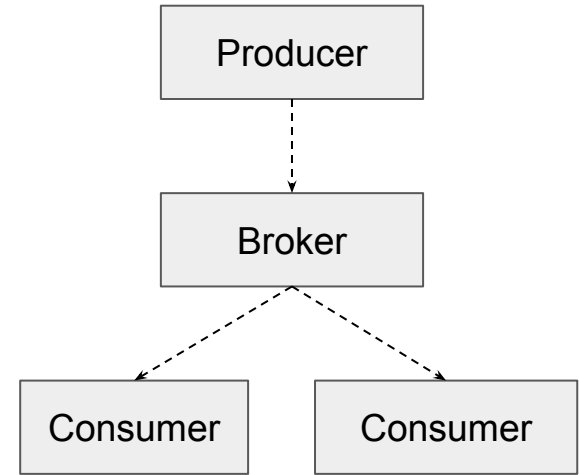
- Server-side architectural patterns continued
- Data and scalability
- Kubernetes
- Load and stress testing

Server-side Architectural Patterns

- Event-driven architecture
- Microkernel / plugin architecture
- Space-based architecture

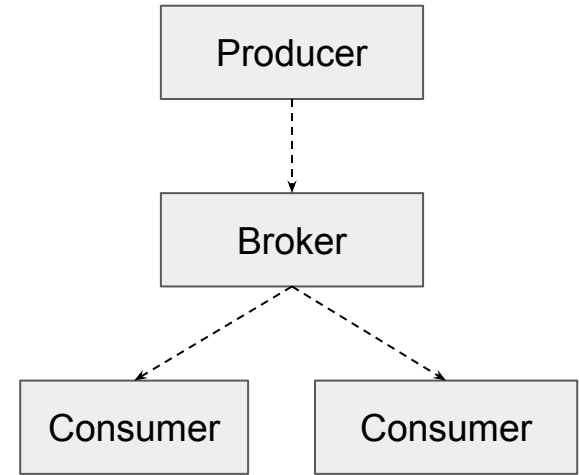
Event-driven Architecture

- When the state of an application changes (i.e. an *event* happens), information about the change is published
- Typically realized using a publish / subscribe (Pub/Sub) pattern
 - Client (subscriber) subscribes to one or more topics of interest by making a request to the server
 - Server (publisher) defines available topics. When new data arrives (is *produced*) for a topic, data is sent to all clients subscribed to the topic
- Needs a message broker (a service for passing and storing messages): Plenty of existing platforms (e.g. Kafka)



Event-driven Architecture

- When the state of an application changes (i.e. an *event* happens), information about the change is published
- Typically realized using a publish / subscribe (Pub/Sub) pattern
 - Client (subscriber) subscribes to one or more topics of interest by making a request to the server
 - Server (publisher) defines available topics. When new data arrives (is *produced*) for a topic, data is sent to all clients subscribed to the topic
- Needs a message broker (a service for passing and storing messages): Plenty of existing platforms (e.g. Kafka)



Check out also Pipes and Filters pattern

Microkernel / plugin architecture

Heiser, G. and Elphinstone, K., 2016. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems (TOCS)*, 34(1), pp.1-29.

Microkernel / plugin architecture

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel

Heiser, G. and Elphinstone, K., 2016. L4 microkernels: The lessons from 20 years of research and deployment. ACM Transactions on Computer Systems (TOCS), 34(1), pp.1-29.

Microkernel / plugin architecture

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel

Heiser, G. and Elphinstone, K., 2016. L4 microkernels: The lessons from 20 years of research and deployment. ACM Transactions on Computer Systems (TOCS), 34(1), pp.1-29.

Microkernel / plugin architecture

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel

*not part of
the kernel*



Microkernel / plugin architecture

Heiser, G. and Elphinstone, K., 2016. L4 microkernels: The lessons from 20 years of research and deployment. ACM Transactions on Computer Systems (TOCS), 34(1), pp.1-29.

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel
 - A small core kernel could be easier to maintain and port to new platforms (architectures)

*not part of
the kernel*



Microkernel / plugin architecture

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel
 - A small core kernel could be easier to maintain and port to new platforms (architectures)
 - Porting the small kernel to new architectures would allow reusing existing user-space programs on new those ports with minimal effort

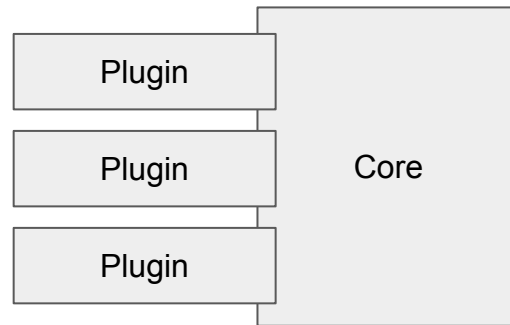
*not part of
the kernel*



Microkernel / plugin architecture

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel
 - A small core kernel could be easier to maintain and port to new platforms (architectures)
 - Porting the small kernel to new architectures would allow reusing existing user-space programs on new those ports with minimal effort

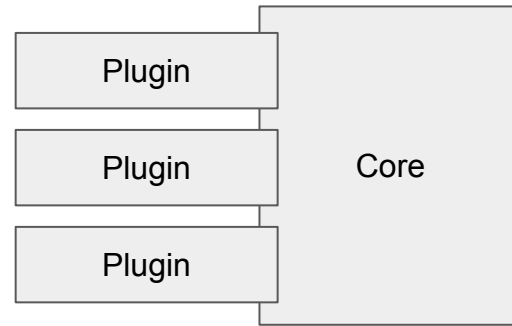
*not part of
the kernel*



Microkernel / plugin architecture

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel
 - A small core kernel could be easier to maintain and port to new platforms (architectures)
 - Porting the small kernel to new architectures would allow reusing existing user-space programs on new those ports with minimal effort
- Idea currently present in virtual machine monitors, virtual machines, and containers (e.g. Linux containers, Docker, ...)

not part of
the kernel



Microkernel / plugin architecture

Heiser, G. and Elphinstone, K., 2016. L4 microkernels: The lessons from 20 years of research and deployment. ACM Transactions on Computer Systems (TOCS), 34(1), pp.1-29.

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel
 - A small core kernel could be easier to maintain and port to new platforms (architectures)
 - Porting the small kernel to new architectures would allow reusing existing user-space programs on new those ports with minimal effort
- Idea currently present in virtual machine monitors, virtual machines, and containers (e.g. Linux containers, Docker, ...)

not part of
the kernel



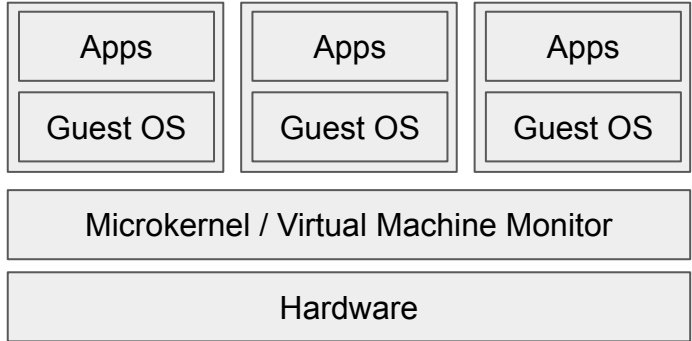
Are Virtual Machine Monitors Microkernels Done Right? –
https://www.usenix.org/legacy/event/hotos05/final_papers_backup/hand/hand_html/index.html

Heiser, G. and Elphinstone, K., 2016. L4 microkernels: The lessons from 20 years of research and deployment. ACM Transactions on Computer Systems (TOCS), 34(1), pp.1-29.

Microkernel / plugin architecture

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel
 - A small core kernel could be easier to maintain and port to new platforms (architectures)
 - Porting the small kernel to new architectures would allow reusing existing user-space programs on new those ports with minimal effort
- Idea currently present in virtual machine monitors, virtual machines, and containers (e.g. Linux containers, Docker, ...)

not part of the kernel



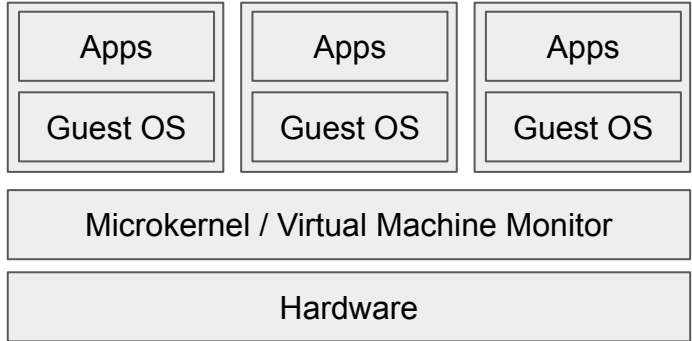
Are Virtual Machine Monitors Microkernels Done Right? – https://www.usenix.org/legacy/event/hotos05/final_papers_backup/hand/hand_html/index.html

Heiser, G. and Elphinstone, K., 2016. L4 microkernels: The lessons from 20 years of research and deployment. ACM Transactions on Computer Systems (TOCS), 34(1), pp.1-29.

Microkernel / plugin architecture

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel
 - A small core kernel could be easier to maintain and port to new platforms (architectures)
 - Porting the small kernel to new architectures would allow reusing existing user-space programs on new those ports with minimal effort
- Idea currently present in virtual machine monitors, virtual machines, and containers (e.g. Linux containers, Docker, ...)

not part of the kernel



Building our apps, we know that we can run them on different platforms

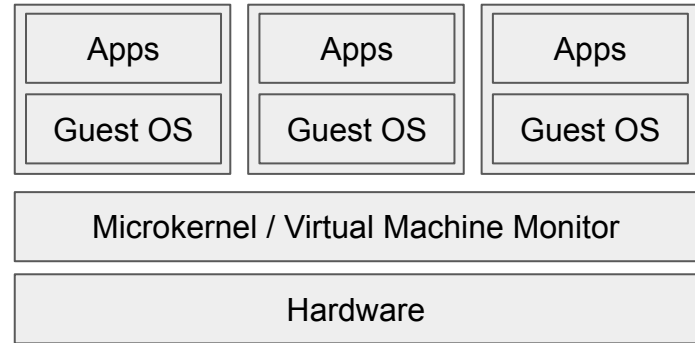
Are Virtual Machine Monitors Microkernels Done Right? – https://www.usenix.org/legacy/event/hotos05/final_papers_backup/hand/hand_html/index.html

Microkernel / plugin architecture

- Origins decades ago (first works in 1960s): evolving hardware, new devices, and so on. Drivers etc traditionally included into a (monolithic) operating system kernel
 - Objective: Implement drivers etc as user-space programs instead, keeping a small core kernel
 - A small core kernel could be easier to maintain and port to new platforms (architectures)
 - Porting the small kernel to new architectures would allow reusing existing user-space programs on new those ports with minimal effort
- Idea currently present in virtual machine monitors, virtual machines, and containers (e.g. Linux containers, Docker, ...)

not part of the kernel

Present also e.g. in browsers, IDEs, etc ("plugin architecture")



Building our apps, we know that we can run them on different platforms

Space-based architecture

Space-based architecture

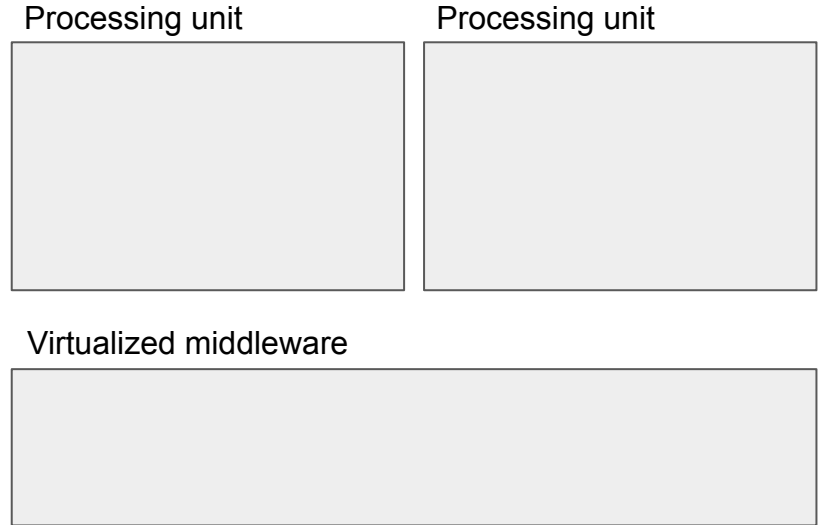
- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)

Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware

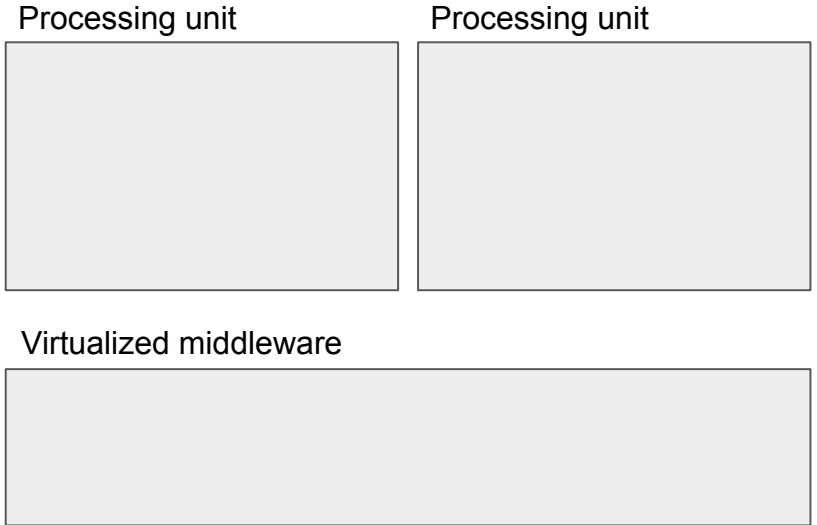
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware



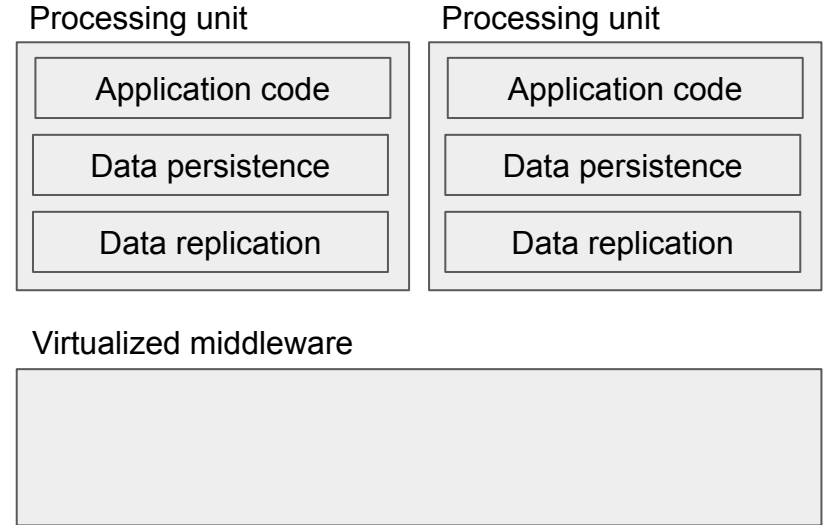
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality



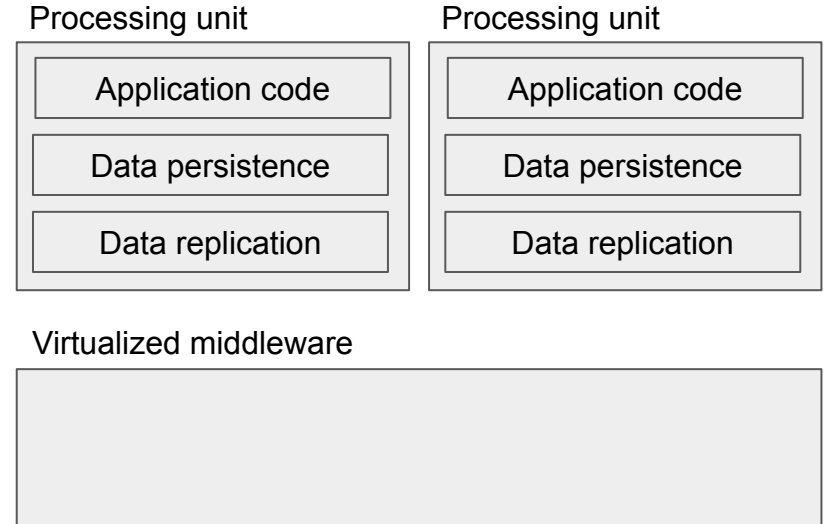
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality



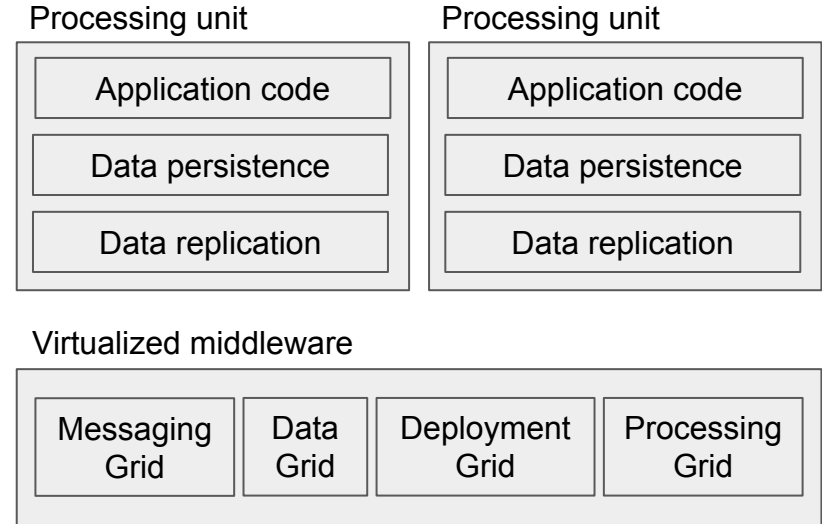
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



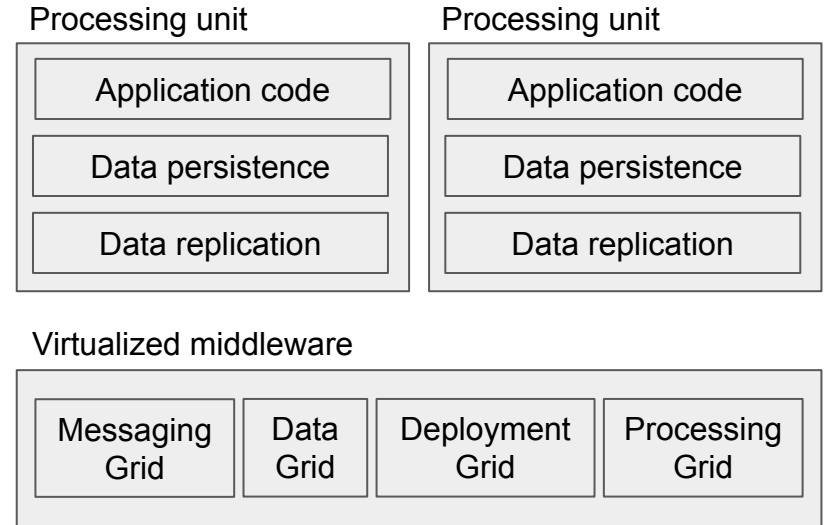
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



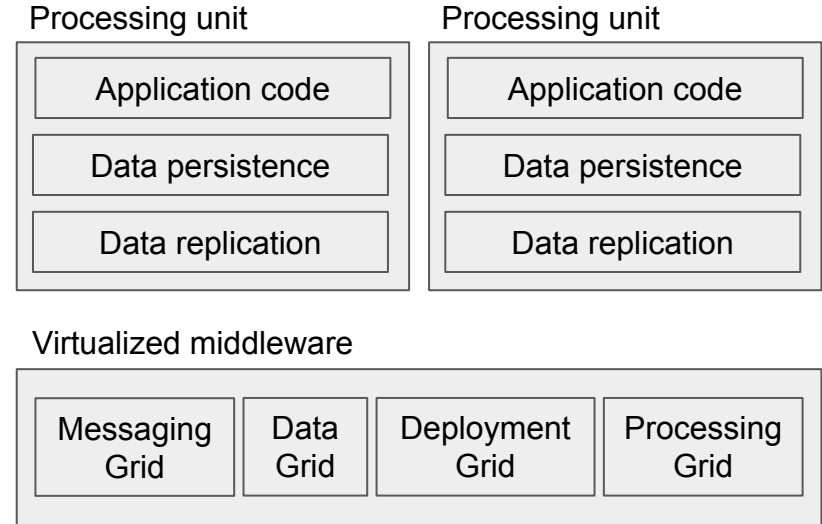
Software Architecture Patterns Chapter 5. Space-Based Architecture:

<https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch05.html>

Grid: potentially multiple connected computers used for handling a task

Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components

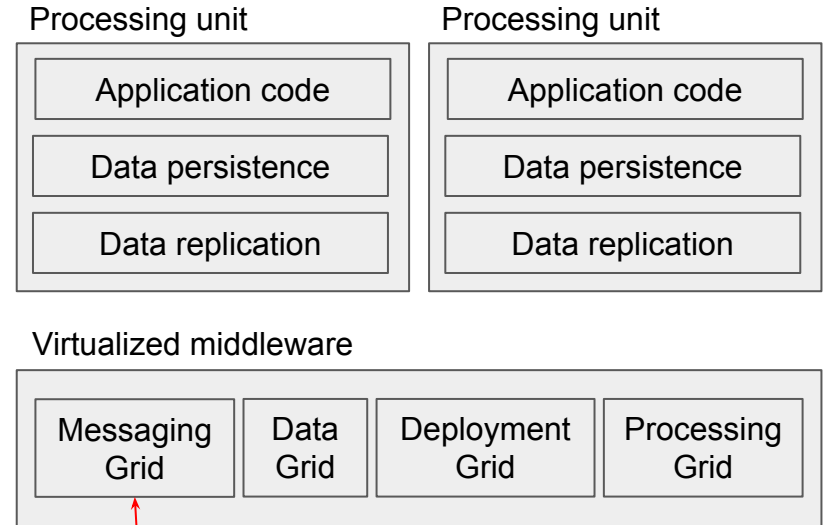


Request

Grid: potentially multiple connected computers used for handling a task

Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components

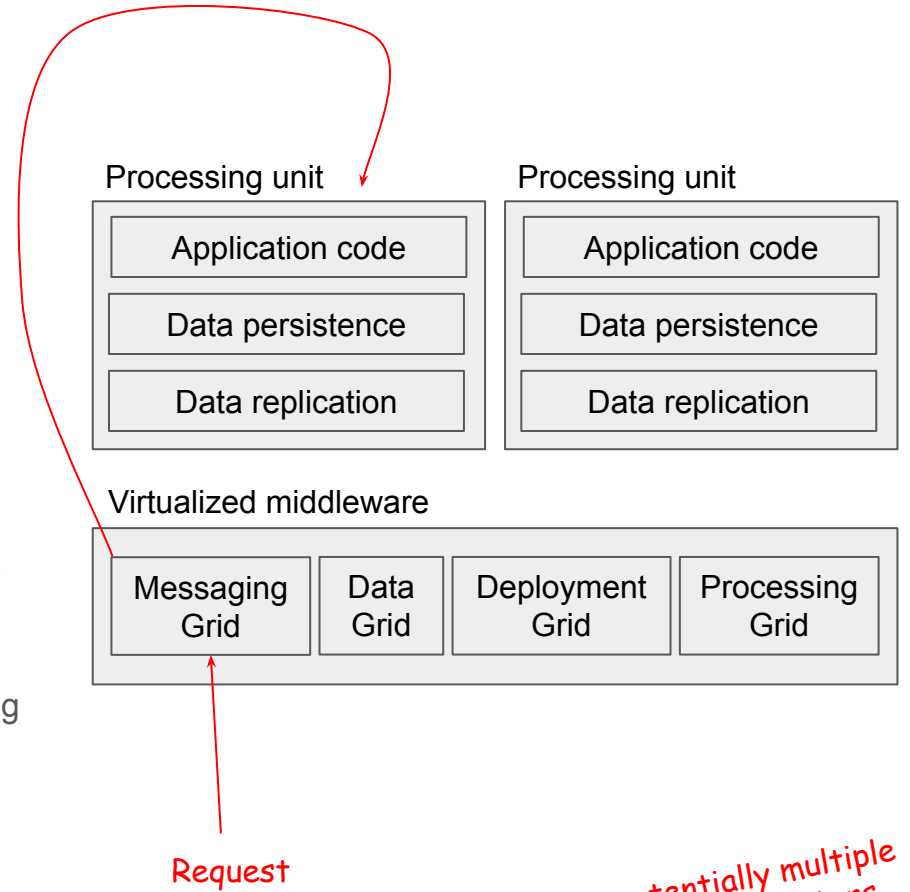


Request

Grid: potentially multiple connected computers used for handling a task

Space-based architecture

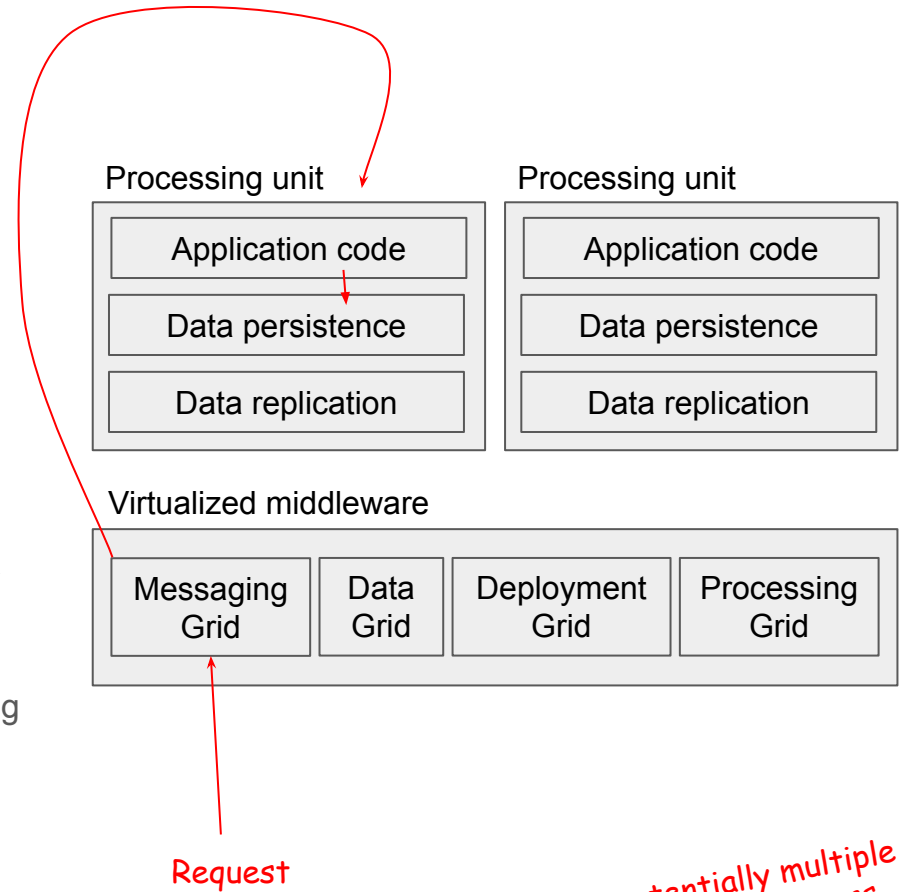
- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



Grid: potentially multiple connected computers used for handling a task

Space-based architecture

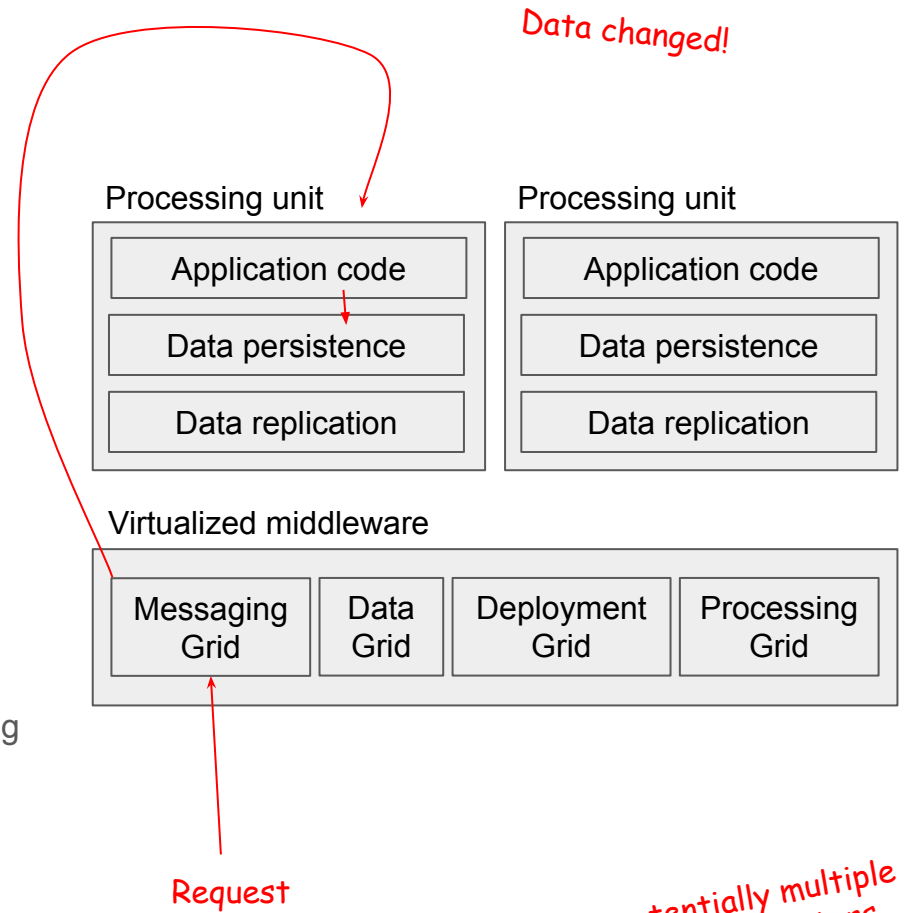
- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



Grid: potentially multiple connected computers used for handling a task

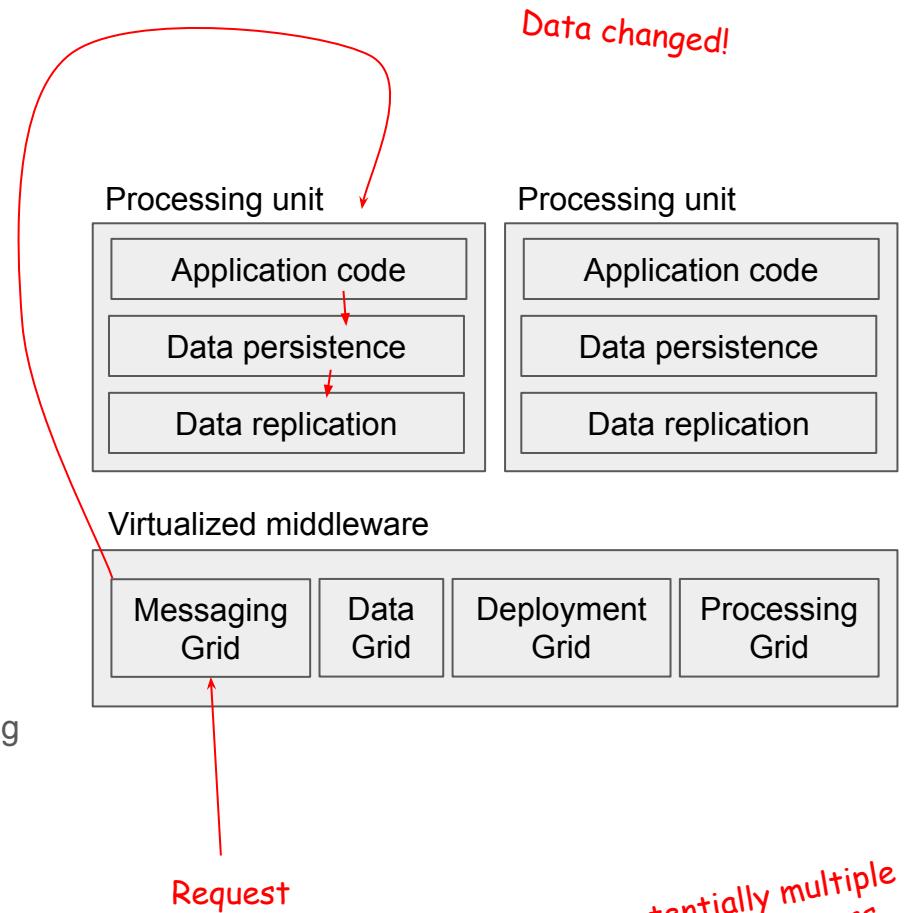
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



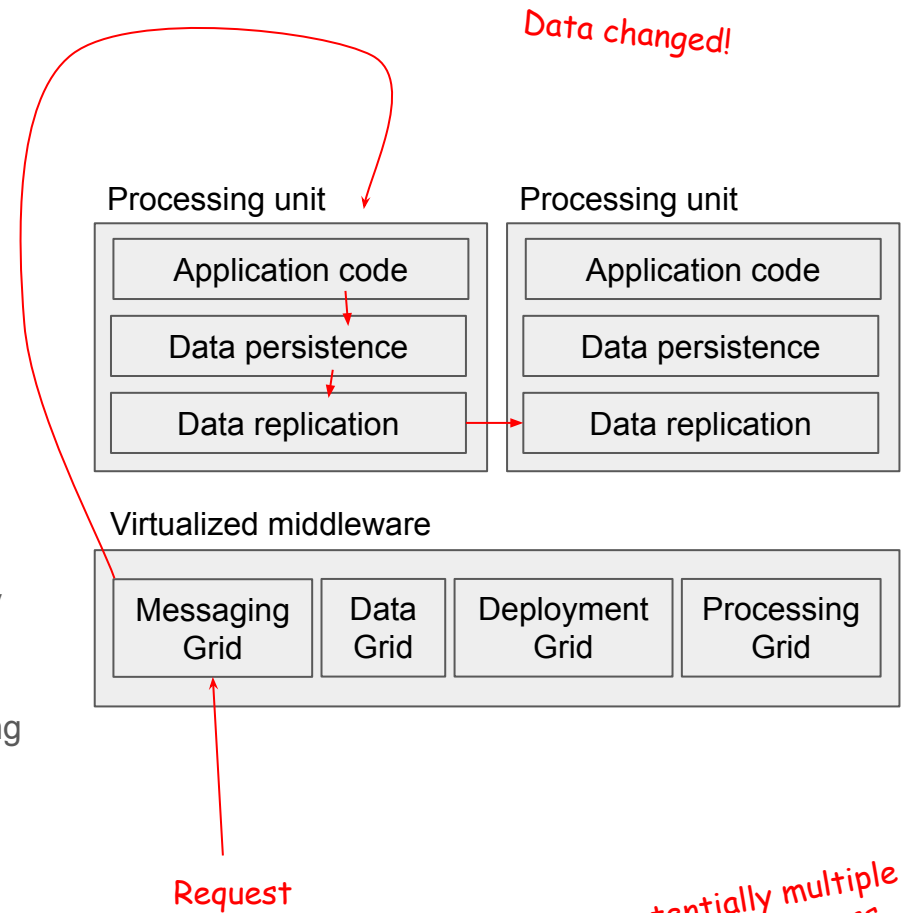
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



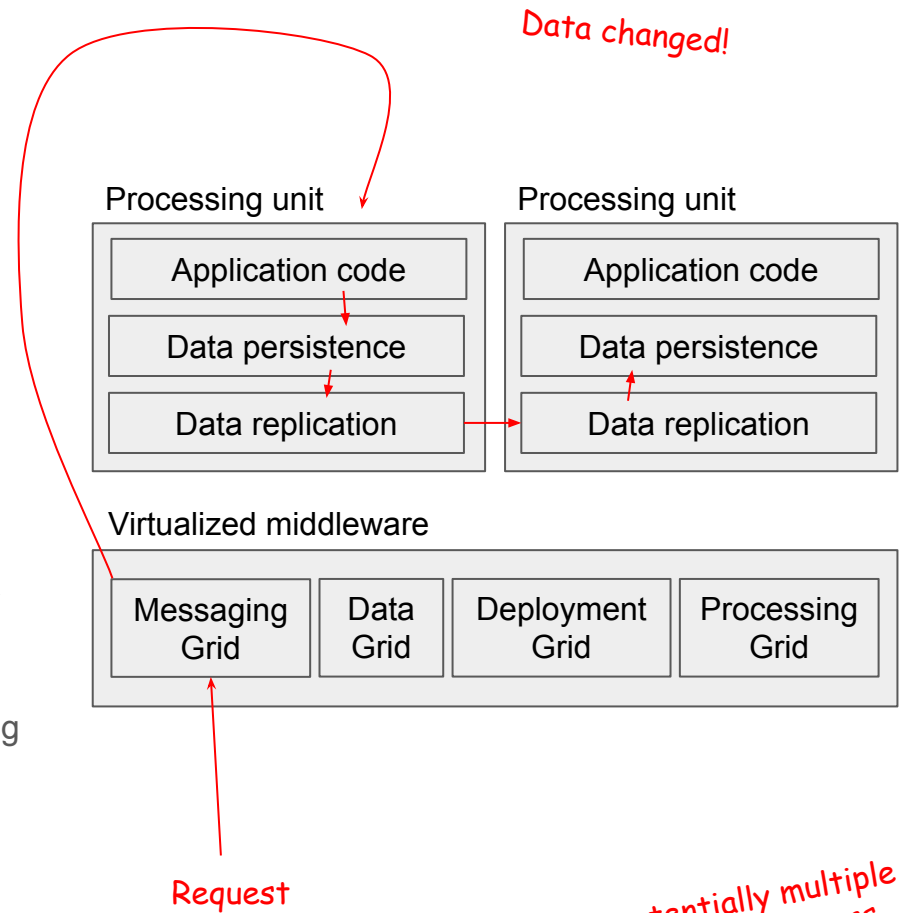
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



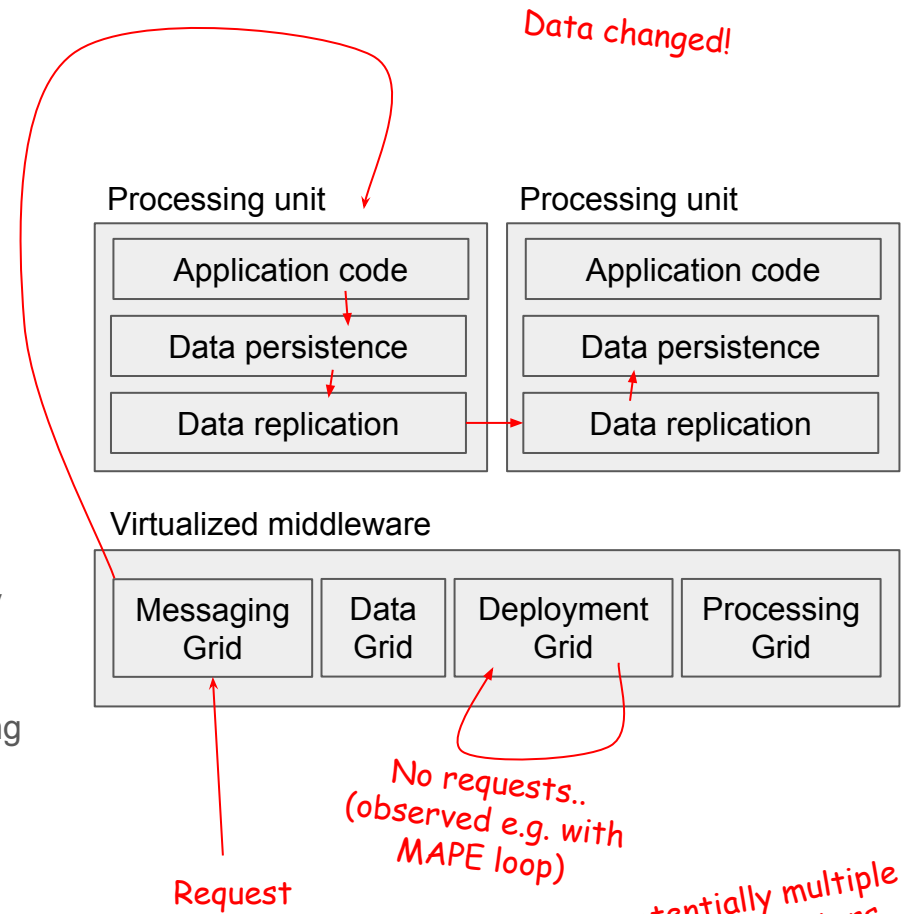
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



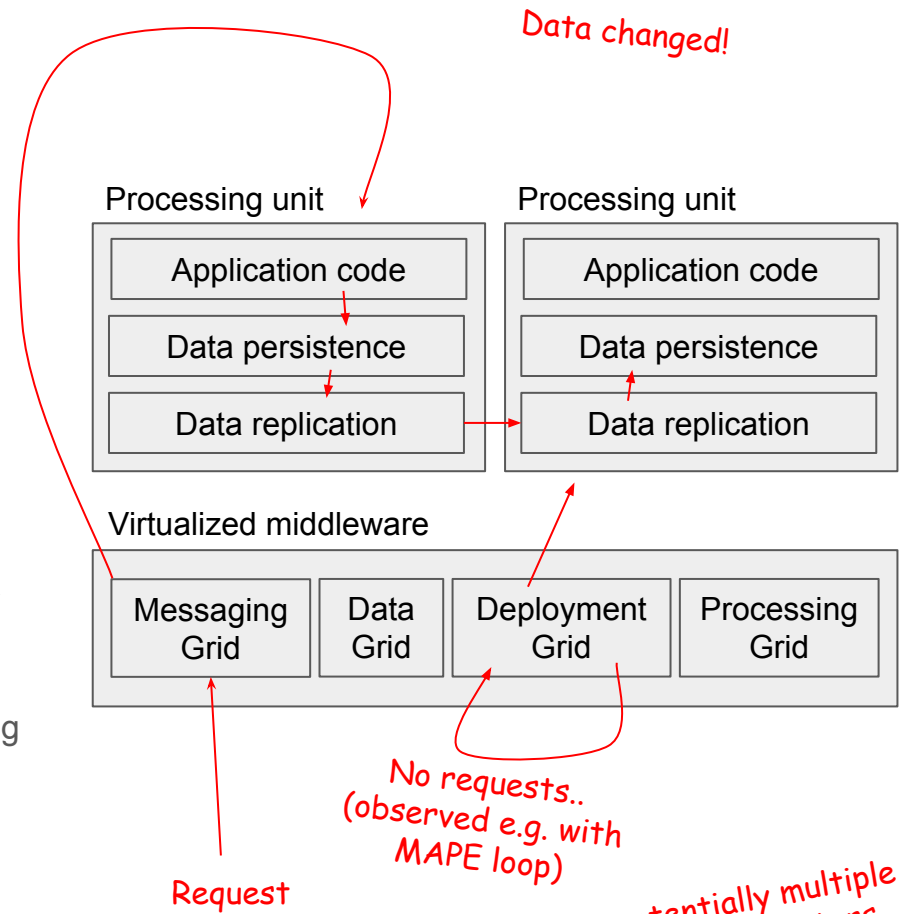
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



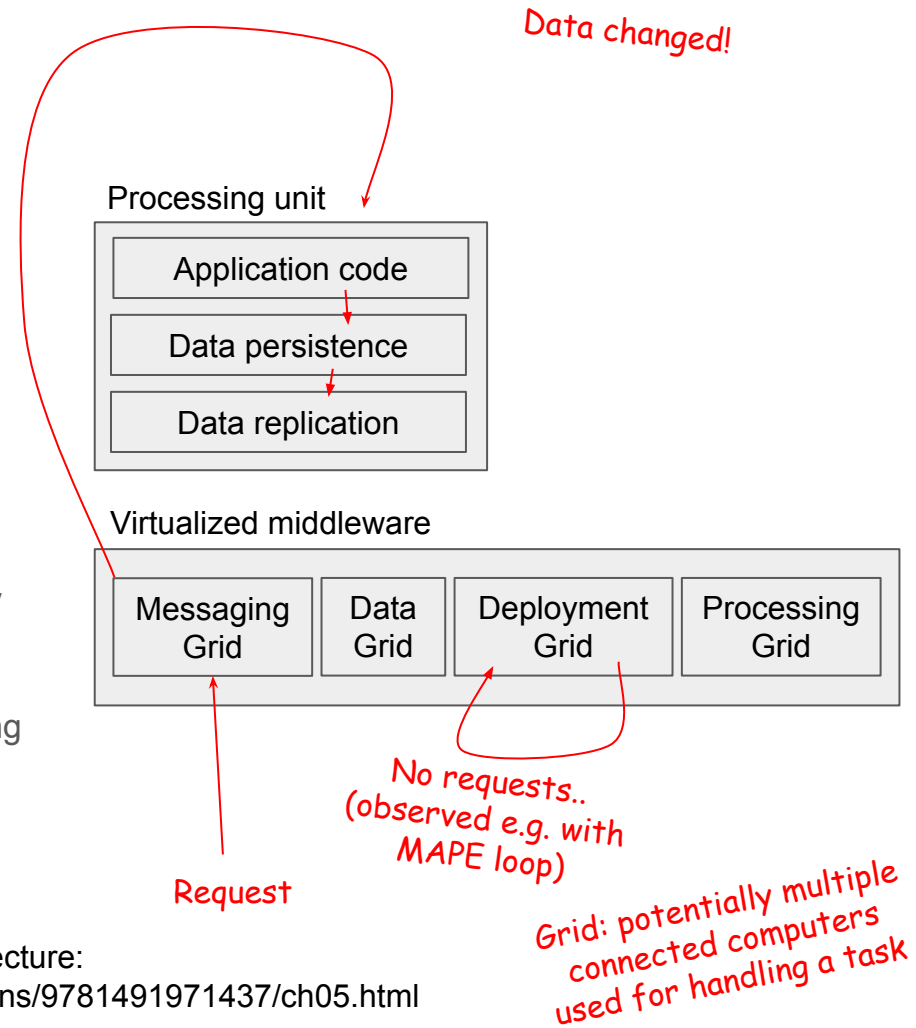
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



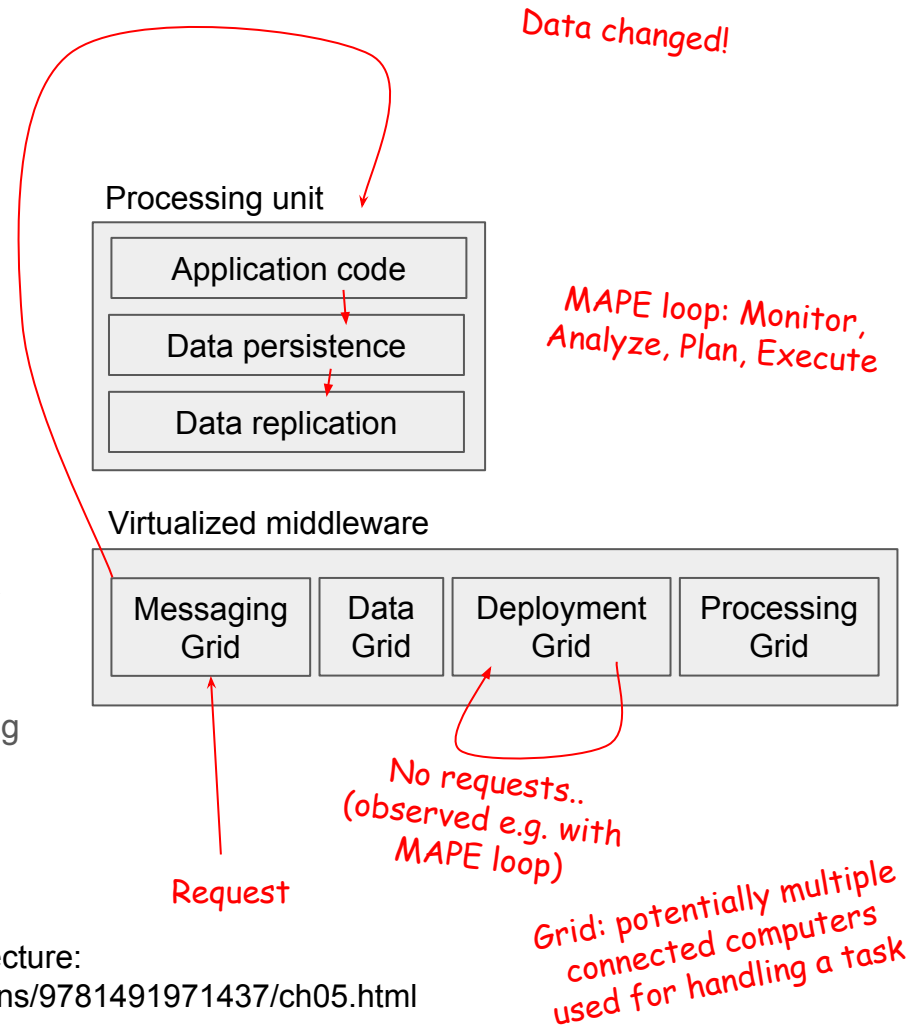
Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



Space-based architecture

- Origins: tuple space and distributed shared memory (memory architecture allowing the use of separated memory units through a single address space)
- Key components: processing units and virtualized middleware
- Processing unit(s) contain application code, data persistence functionality, data replication functionality
- Virtualized middleware used to direct requests to processing units, to replicate data between processing units, to start and stop processing units based on demand, and to (potentially) distribute processing if the application has been divided into multiple components



Space-based architecture

Space-based architecture

- Requests can be directed to any processing units → data must be up to date between processing units

Space-based architecture

- Requests can be directed to any processing units → data must be up to date between processing units
- Data grid responsible for maintaining up-to-date data in each processing unit through their data replication engines (but, data replication engines can and also interact with each others)

Space-based architecture

- Requests can be directed to any processing units → data must be up to date between processing units
- Data grid responsible for maintaining up-to-date data in each processing unit through their data replication engines (but, data replication engines can and also interact with each others)
- Applications can also be divided into further processing units (~microservices) → processing grid takes some responsibility of sharing requests to correct processing units

Space-based architecture

- Requests can be directed to any processing units → data must be up to date between processing units
- Data grid responsible for maintaining up-to-date data in each processing unit through their data replication engines (but, data replication engines can and also interact with each others)
- Applications can also be divided into further processing units (~microservices) → processing grid takes some responsibility of sharing requests to correct processing units

See also...

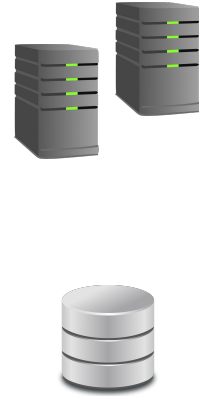
*Designing a Scalable Twitter (2009):
https://natishalom.typepad.com/nati_shaloms_blog/2009/04/writing-your-own-scalable-twitter.html*

*The Case for Shared Nothing (1985):
<https://dsf.berkeley.edu/papers/hpts85-nothing.pdf>*

Data and Scalability

Data as a bottleneck

- Horizontal scalability = scaling out = increasing number of servers



Data as a bottleneck

- Horizontal scalability = scaling out = increasing number of servers



Data as a bottleneck

- Horizontal scalability = scaling out = increasing number of servers



Data as a bottleneck

- Horizontal scalability = scaling out = increasing number of servers



Data as a bottleneck

- Horizontal scalability = scaling out = increasing number of servers
- Need to do something with database or it cannot handle the load



ACID, CAP, BASE

ACID, CAP, BASE

- ACID

- Database transaction properties for guaranteeing data consistency
- **A**tomicity – each transaction (a set of database operations) must complete fully or not complete at all
- **C**onsistency – each transaction preserves the consistency of the database
- **I**solation – each transaction must be hidden from other transactions running in parallel
- **D**urability – once a transaction has been completed, the system must guarantee that the transaction outcomes persist any subsequent system malfunctions

Haerder, Theo, and Andreas Reuter. "Principles of transaction-oriented database recovery." ACM computing surveys (CSUR) 15.4 (1983): 287-317.

ACID, CAP, BASE

- ACID

- Database transaction properties for guaranteeing data consistency
- **A**tomicity – each transaction (a set of database operations) must complete fully or not complete at all
- **C**onsistency – each transaction preserves the consistency of the database
- **I**solation – each transaction must be hidden from other transactions running in parallel
- **D**urability – once a transaction has been completed, the system must guarantee that the transaction outcomes persist any subsequent system malfunctions

- CAP Theorem

- Any distributed data store can have at most two out of (1) **C**onsistency, (2) **A**vailability, and (3) **P**artition tolerance
- **C**onsistency – data read is up to date and reflects the latest write
- **A**vailability – every request receives a response
- **P**artition tolerance – when data is distributed over multiple systems (or multiple partitions), the database works even if messaging between systems fails

Haerder, Theo, and Andreas Reuter. "Principles of transaction-oriented database recovery." ACM computing surveys (CSUR) 15.4 (1983): 287-317.

Fox, Armando, and Eric A. Brewer. "Harvest, yield, and scalable tolerant systems." Proceedings of the Seventh Workshop on Hot Topics in Operating Systems. IEEE, 1999.

ACID, CAP, BASE

- ACID

- Database transaction properties for guaranteeing data consistency
- **A**tomicity – each transaction (a set of database operations) must complete fully or not complete at all
- **C**onsistency – each transaction preserves the consistency of the database
- **I**solation – each transaction must be hidden from other transactions running in parallel
- **D**urability – once a transaction has been completed, the system must guarantee that the transaction outcomes persist any subsequent system malfunctions

- CAP Theorem

- Any distributed data store can have at most two out of (1) **C**onsistency, (2) **A**vailability, and (3) **P**artition tolerance
- **C**onsistency – data read is up to date and reflects the latest write
- **A**vailability – every request receives a response
- **P**artition tolerance – when data is distributed over multiple systems (or multiple partitions), the database works even if messaging between systems fails

- BASE

- Accept that reaching consistency may take time, accept partial failures
- **B**asically **A**vailable – distribute data and accept failures → at least some users will get data
- **S**oft state – state information can be lost e.g. in system crashes or with network issues
- **E**ventually consistent – data will be consistent at some point in time

Haerder, Theo, and Andreas Reuter. "Principles of transaction-oriented database recovery." ACM computing surveys (CSUR) 15.4 (1983): 287-317.

Fox, Armando, and Eric A. Brewer. "Harvest, yield, and scalable tolerant systems." Proceedings of the Seventh Workshop on Hot Topics in Operating Systems. IEEE, 1999.

Pritchett, Dan. "BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability." ACM Queue 6.3 (2008): 48-55.

ACID, CAP, BASE

- ACID

- Database transaction properties for guaranteeing data consistency
- **A**tomicity – each transaction (a set of database operations) must complete fully or not complete at all
- **C**onsistency – each transaction preserves the consistency of the database
- **I**solation – each transaction must be hidden from other transactions running in parallel
- **D**urability – once a transaction has been completed, the system must guarantee that the transaction outcomes persist any subsequent system malfunctions

- CAP Theorem

- Any distributed data store can have at most two out of (1) **C**onsistency, (2) **A**vailability, and (3) **P**artition tolerance
- **C**onsistency – data read is up to date and reflects the latest write
- **A**vailability – every request receives a response
- **P**artition tolerance – when data is distributed over multiple systems (or multiple partitions), the database works even if messaging between systems fails

- BASE

- Accept that reaching consistency may take time, accept partial failures
- **B**asically **A**vailable – distribute data and accept failures → at least some users will get data
- **S**oft state – state information can be lost e.g. in system crashes or with network issues
- **E**ventually consistent – data will be consistent at some point in time

Haerder, Theo, and Andreas Reuter. "Principles of transaction-oriented database recovery." *ACM computing surveys (CSUR)* 15.4 (1983): 287-317.

Fox, Armando, and Eric A. Brewer. "Harvest, yield, and scalable tolerant systems." *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. IEEE, 1999.

Event-Driven Architecture suggested as a way to inform users when state has become consistent



Pritchett, Dan. "BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability." *ACM Queue* 6.3 (2008): 48-55.

About CAP Theorem...

CAP Twelve Years Later: How the "Rules" Have Changed

<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/>

About CAP Theorem...

- Network partitions do happen every now and then, so the choice is between AP and CP

CAP Twelve Years Later: How the "Rules" Have Changed

<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/>

About CAP Theorem...

- Network partitions do happen every now and then, so the choice is between AP and CP
 - Or is it?

CAP Twelve Years Later: How the "Rules" Have Changed

<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/>

About CAP Theorem...

- Network partitions do happen every now and then, so the choice is between AP and CP
 - Or is it?
 - Network partitions highly unlikely (given proper infrastructure) – other problems can also happen (e.g. broken hardware) – one question is how to handle these.

CAP Twelve Years Later: How the "Rules" Have Changed

<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/>

About CAP Theorem...

- Network partitions do happen every now and then, so the choice is between AP and CP
 - Or is it?
 - Network partitions highly unlikely (given proper infrastructure) – other problems can also happen (e.g. broken hardware) – one question is how to handle these.
 - Another question is how to mitigate network partitions.

CAP Twelve Years Later: How the "Rules" Have Changed

<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/>

About CAP Theorem...

- Network partitions do happen every now and then, so the choice is between AP and CP
 - Or is it?
 - Network partitions highly unlikely (given proper infrastructure) – other problems can also happen (e.g. broken hardware) – one question is how to handle these.
 - Another question is how to mitigate network partitions.
- In practice, the big question is “how to achieve effectively CA” in large systems.

CAP Twelve Years Later: How the "Rules" Have Changed

<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/>

About CAP Theorem...

- Network partitions do happen every now and then, so the choice is between AP and CP
 - Or is it?
 - Network partitions highly unlikely (given proper infrastructure) – other problems can also happen (e.g. broken hardware) – one question is how to handle these.
 - Another question is how to mitigate network partitions.
- In practice, the big question is “how to achieve effectively CA” in large systems.

*E.g. Google Spanner:
Brewer, E., 2017. Spanner,
TrueTime and the CAP theorem.*

CAP Twelve Years Later: How the "Rules" Have Changed


<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/>

Practical database scaling

Practical database scaling


- Caching query results
- Effective indexing
- Read replication and sharding

Practical database scaling

- Caching query results 
- Effective indexing
- Read replication and sharding

Cache query results, flush cache only when data in database changes → fewer database queries

Practical database scaling

- Caching query results 
- Effective indexing
- Read replication and sharding

*Cache query results, flush cache only when data
in database changes → fewer database queries
Works when #reads >> #writes*



Practical database scaling

- Caching query results
- Effective indexing
- Read replication and sharding

Cache query results, flush cache only when data in database changes → fewer database queries
Works when #reads \gg #writes

Have query indexes in place → faster queries (index lookups vs table scans)

Practical database scaling

- Caching query results 
- Effective indexing 
- Read replication and sharding

*Cache query results, flush cache only when data in database changes → fewer database queries
Works when #reads >> #writes*

*Have query indexes in place → faster queries
(index lookups vs table scans)*

...

Read replication and sharding



Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding



Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data



Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data



Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers



Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers



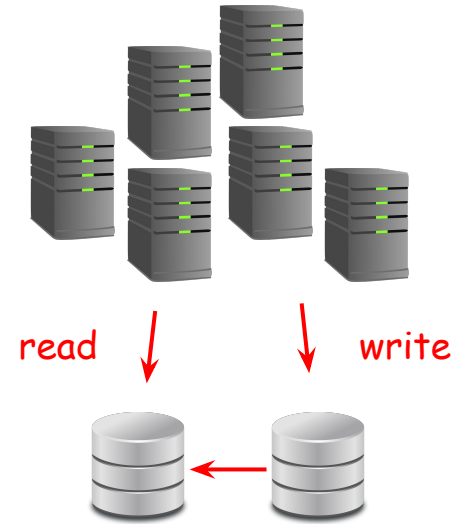
Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers
 - Read replication: one database dedicated for handling writes, others for reads



Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers
 - Read replication: one database dedicated for handling writes, others for reads



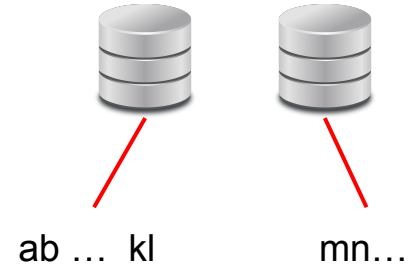
Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers
 - Read replication: one database dedicated for handling writes, others for reads
- Data sharding: Data divided into shards that are stored in two or more servers



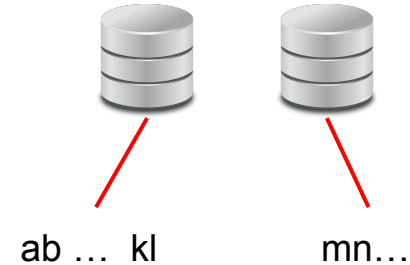
Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers
 - Read replication: one database dedicated for handling writes, others for reads
- Data sharding: Data divided into shards that are stored in two or more servers



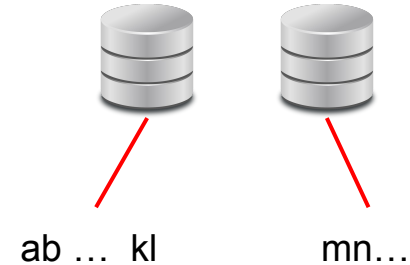
Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers
 - Read replication: one database dedicated for handling writes, others for reads
- Data sharding: Data divided into shards that are stored in two or more servers
 - Consistency in place
 - Queries that require data from multiple servers can be slower
 - On the other hand, queries requiring data from a single server can be (in some cases) faster



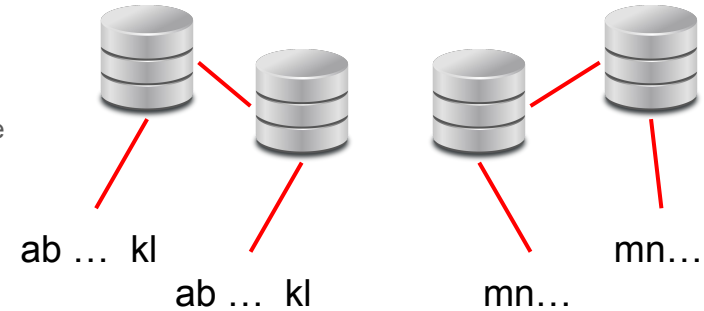
Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers
 - Read replication: one database dedicated for handling writes, others for reads
- Data sharding: Data divided into shards that are stored in two or more servers
 - Consistency in place
 - Queries that require data from multiple servers can be slower
 - On the other hand, queries requiring data from a single server can be (in some cases) faster
- Possibility to also do both replication and sharding



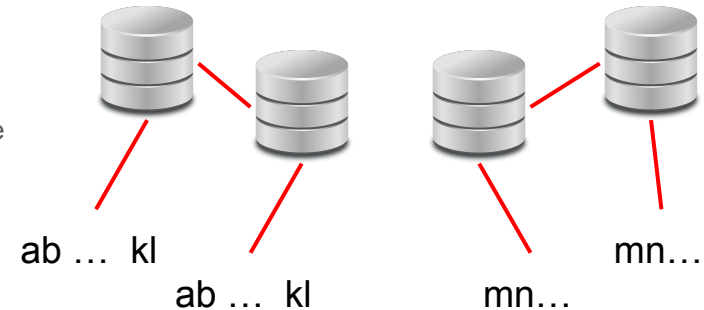
Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers
 - Read replication: one database dedicated for handling writes, others for reads
- Data sharding: Data divided into shards that are stored in two or more servers
 - Consistency in place
 - Queries that require data from multiple servers can be slower
 - On the other hand, queries requiring data from a single server can be (in some cases) faster
- Possibility to also do both replication and sharding



Read replication and sharding

- Two ways to distribute database data: (1) data replication, (2) data sharding
- Data replication: entire data is replicated over two or more servers, both maintaining copies of the data
 - Higher availability
 - To achieve consistency, data needs to be updated between servers
 - Read replication: one database dedicated for handling writes, others for reads
- Data sharding: Data divided into shards that are stored in two or more servers
 - Consistency in place
 - Queries that require data from multiple servers can be slower
 - On the other hand, queries requiring data from a single server can be (in some cases) faster
- Possibility to also do both replication and sharding



Also fragmentation - dividing data into fragments in a database for performance reasons; e.g. storing log data into weekly fragments which would allow faster for specific timeframes.

Updating data between servers

Updating data between servers

- Using consensus protocols (often Paxos) for updating distributed data (also Raft, EPaxos, ...)

Updating data between servers

- Using consensus protocols (often Paxos) for updating distributed data (also Raft, EPaxos, ...)

Marandi, P. J., Primi, M., Schiper, N., & Pedone, F. (2010, June). Ring Paxos: A high-throughput atomic broadcast protocol. In 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN).

Moraru, I., Andersen, D. G., & Kaminsky, M. (2013, November). There is more consensus in egalitarian parliaments. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.

Van Renesse, R., & Altinbuken, D. (2015). Paxos made moderately complex. ACM Computing Surveys (CSUR), 47(3), 1-36.

Rao, J., Shekita, E. J., & Tata, S. (2011). Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. Proceedings of the VLDB Endowment.

Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. In 2014 USENIX Annual Technical Conference (Usenix ATC 14).

etc...

Updating data between servers

*Not going
into these...*

- Using consensus protocols (often Paxos) for updating distributed data (also Raft, EPaxos, ...)

Marandi, P. J., Primi, M., Schiper, N., & Pedone, F. (2010, June). Ring Paxos: A high-throughput atomic broadcast protocol. In 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN).

Moraru, I., Andersen, D. G., & Kaminsky, M. (2013, November). There is more consensus in egalitarian parliaments. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.

Van Renesse, R., & Altinbuken, D. (2015). Paxos made moderately complex. ACM Computing Surveys (CSUR), 47(3), 1-36.

Rao, J., Shekita, E. J., & Tata, S. (2011). Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. Proceedings of the VLDB Endowment.

Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. In 2014 USENIX Annual Technical Conference (Usenix ATC 14).

etc...

Kubernetes

Kubernetes

Kubernetes

- Scaling requires adjustment (of the number) of running applications and directing traffic to the said applications

Kubernetes

- Scaling requires adjustment (of the number) of running applications and directing traffic to the said applications
- “Kubernetes [...] is an open-source system for automating deployment, scaling, and management of containerized applications.” – *Kubernetes.io*

Kubernetes

- Scaling requires adjustment (of the number) of running applications and directing traffic to the said applications
- “Kubernetes [...] is an open-source system for automating deployment, scaling, and management of containerized applications.” – *Kubernetes.io*
- Key terminology:
 - Container: an image containing software and its dependencies (hello Docker!)
 - Pod: a set of running containers
 - Node: a (virtual) machine with functionality needed to run pods
 - Cluster: a group ($n \Rightarrow 1$) of nodes

Kubernetes

Kubernetes

- Large scale deployment in practice requires a bunch of nodes (servers or virtual machines), but...

Kubernetes

- Large scale deployment in practice requires a bunch of nodes (servers or virtual machines), but...
- For learning purposes, there's multiple ways to go
 - Minikube – <https://minikube.sigs.k8s.io/>
 - MicroK8s – <https://microk8s.io/>
 - K3s – <https://k3s.io/> and K3D – <https://k3d.io/> (for running K3s in Docker)
 - Kind – <https://kind.sigs.k8s.io/> (running Kubernetes with Docker containers)
 - ...

Kubernetes

- Large scale deployment in practice requires a bunch of nodes (servers or virtual machines), but...
- For learning purposes, there's multiple ways to go
 - Minikube – <https://minikube.sigs.k8s.io/>
 - MicroK8s – <https://microk8s.io/>
 - K3s – <https://k3s.io/> and K3D – <https://k3d.io/> (for running K3s in Docker)
 - Kind – <https://kind.sigs.k8s.io/> (running Kubernetes with Docker containers)
 - ...

Check out also <https://kubernetes.io/docs/tutorials/kubernetes-basics/>

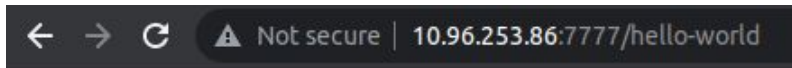
Minikube demo

<https://kubernetes.io/docs/tutorials/hello-minikube/>

<https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-interactive/>

Minikube demo – minikube and kubectl already installed

- Start minikube
 - `minikube start`
- Allow using local docker images in minikube
 - `eval $(minikube -p minikube docker-env)`
- Build a Docker image (In a folder with Dockerfile)
 - `minikube image build -t my-app-image .`
 - For the present demo, we built a web app that exposes port 7777
- Open minikube dashboard (separate terminal)
 - `minikube dashboard`
- Create deployment
 - `kubectl create deployment my-kube-app --image=my-app-image`
- Change image pull policy to pull from local (well, to not pull from global)
 - Adjust deployment config "imagePullPolicy: Always" to "imagePullPolicy: Never"
- Expose pod using a load balancer
 - Create a tunnel (as root, separate terminal):
`minikube tunnel`
 - Create a load balancer service for our app:
`kubectl expose deployment my-kube-app --type=LoadBalancer --port=7777`
- Find load balancer (external) IP:
 - `kubectl get svc`
- Access server at port :)



Requested path /hello-world

Minikube demo – minikube and kubectl already installed

- Updating deployed image
 - Update contents of docker image and rebuild it
 - Remove a pod → you'll notice that the pod will be redeployed → latest image deployed
- Scaling up (by creating replicas)
 - Show details (find app name)
 - `kubectl get all`
 - Scale the deployment to three replicas
 - `kubectl scale deployment.apps/my-kube-app --replicas=3`
- Scaling automatically
 - Enable metrics server (checking metrics for scaling)
 - `minikube addons enable metrics-server`
 - At least 1 replica, at most 5 replicas, scale up if CPU load is over 25%
 - `kubectl autoscale deployment.apps/my-kube-app --min=1 --max=5 --cpu-percent=25`
 - Might need some config trickery to get working locally, e.g. running with
 - `minikube start --extra-config=kubelet.housekeeping-interval=10s`
 - And setting resource limits for container config (to allow counting of CPU usage) ...

```
125 ▾ spec:
126 ▾   containers:
127 ▾     - name: my-app-image
128 ▾       image: my-app-image
129 ▾       resources:
130 ▾         limits:
131 ▾           cpu: 1|
132 ▾
```

Minikube demo – minikube and kubectl already installed

- Normally configuration through config files; as an example format, see
 - `kubectl get deployment -o my-kube-app -o yaml`
- Deployment
 - `kubectl apply -f my-kube-app-deployment.yaml`

Kubernetes and databases?

- Replication and sharding can take some effort to setup – in practice, there are several companies offering scalable databases as a service
- However, pretty much all databases can be used with Kubernetes
 - Some have ready bundles with replication out of box (see e.g. <https://www.kubegres.io/>)
 - Others have tutorials to get started (see e.g. <https://www.mongodb.com/docs/kubernetes-operator/master/tutorial/deploy-replica-set/>)

Load and stress testing

Load and stress testing

Load and stress testing

- Previously looked into performance testing
 - What is the performance of the application? How fast do the different endpoints respond to requests?

Load and stress testing

- Previously looked into performance testing
 - What is the performance of the application? How fast do the different endpoints respond to requests?
- There are other types of testing approaches
 - Load testing: testing that the application can handle the expected load
 - Stress testing: testing how the application fares in extreme conditions



Load and stress testing

- Previously looked into performance testing
 - What is the performance of the application? How fast do the different endpoints respond to requests?
- There are other types of testing approaches
 - Load testing: testing that the application can handle the expected load
 - Stress testing: testing how the application fares in extreme conditions

"The usual day"



Load and stress testing

- Previously looked into performance testing
 - What is the performance of the application? How fast do the different endpoints respond to requests?
- There are other types of testing approaches
 - Load testing: testing that the application can handle the expected load 
 - Stress testing: testing how the application fares in extreme conditions 

Load testing - k6

- The usual day: *there are some 25 users making one request per second, growing up to 100 users, and then declining to 50 users.*
- Expectations:
 - 99.9% of the requests should be handled in under 300 milliseconds and
 - less than 0.1% of the requests should have errors

Load testing - k6

- The usual day: *there are some 25 users making one request per second, growing up to 100 users, and then declining to 50 users.*
- Expectations:
 - 99.9% of the requests should be handled in under 300 milliseconds and
 - less than 0.1% of the requests should have errors

```
import http from "k6/http";
import { sleep } from 'k6';

export const options = {
  stages: [
    { duration: "30s", target: 25 },
    { duration: "1m", target: 100 },
    { duration: "1m", target: 50 },
  ],
  thresholds: {
    http_req_duration: ["p(99.9)<300"],
    http_req_failed: ["rate<0.001"]
  },
};

export default function () {
  http.get("my-address");
  sleep(1);
}
```

Load testing - k6

Typically tests are run over a longer time period



- The usual day: *there are some 25 users making one request per second, growing up to 100 users, and then declining to 50 users.*
- Expectations:
 - 99.9% of the requests should be handled in under 300 milliseconds and
 - less than 0.1% of the requests should have errors

```
import http from "k6/http";
import { sleep } from 'k6';

export const options = {
  stages: [
    { duration: "30s", target: 25 },
    { duration: "1m", target: 100 },
    { duration: "1m", target: 50 },
  ],
  thresholds: {
    http_req_duration: ["p(99.9)<300"],
    http_req_failed: ["rate<0.001"]
  },
};

export default function () {
  http.get("my-address");
  sleep(1);
}
```


Load testing - k6

Typically tests are run over a longer time period



- The usual day: *there are some 25 users making one request per second, growing up to 100 users, and then declining to 50 users.*
- Expectations:
 - 99.9% of the requests should be handled in under 300 milliseconds and
 - less than 0.1% of the requests should have errors

Can and should test out most important endpoints and use cases



```
import http from "k6/http";
import { sleep } from 'k6';

export const options = {
  stages: [
    { duration: "30s", target: 25 },
    { duration: "1m", target: 100 },
    { duration: "1m", target: 50 },
  ],
  thresholds: {
    http_req_duration: ["p(99.9)<300"],
    http_req_failed: ["rate<0.001"]
  },
};

export default function () {
  http.get("my-address");
  sleep(1);
}
```

Stress testing - k6

- Not so usual day: *there are some 2500 users making one request per second, growing up to 10000 users, and then declining to 500 users.*
- Same expectations:
 - 99.9% of the requests should be handled in under 300 milliseconds and
 - less than 0.1% of the requests should have errors

Stress testing - k6

- Not so usual day: *there are some 2500 users making one request per second, growing up to 10000 users, and then declining to 500 users.*
- Same expectations:
 - 99.9% of the requests should be handled in under 300 milliseconds and
 - less than 0.1% of the requests should have errors

```
import http from "k6/http";
import { sleep } from 'k6';

export const options = {
  stages: [
    { duration: "2m", target: 2500 },
    { duration: "5m", target: 10000 },
    { duration: "2m", target: 500 },
  ],
  thresholds: {
    http_req_duration: ["p(99.9)<300"],
    http_req_failed: ["rate<0.001"]
  },
};

export default function () {
  http.get("my-address");
  sleep(1);
}
```


Stress testing - k6

- Not so usual day: *there are some 2500 users making one request per second, growing up to 10000 users, and then declining to 500 users.*
- Same expectations:
 - 99.9% of the requests should be handled in under 300 milliseconds and
 - less than 0.1% of the requests should have errors

Stress testing from a single computer? Consider cloud services such as k6, supervisor, etc ...

```
import http from "k6/http";
import { sleep } from 'k6';

export const options = {
  stages: [
    { duration: "2m", target: 2500 },
    { duration: "5m", target: 10000 },
    { duration: "2m", target: 500 },
  ],
  thresholds: {
    http_req_duration: ["p(99.9)<300"],
    http_req_failed: ["rate<0.001"]
  },
};

export default function () {
  http.get("my-address");
  sleep(1);
}
```