Università della Svizzera italiana | Faculty of Informatics

Architectural Patterns (2)

Prof. Cesare Pautasso
http://www.pautasso.info
cesare.pautasso@usi.ch
@pautasso

Event Monitor

Goal: inform clients about events happening at the service

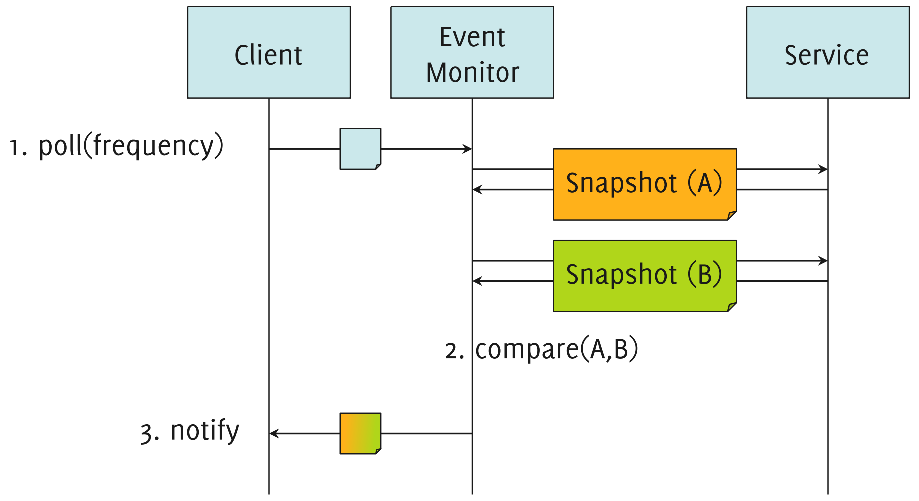
poll and compare state snapshots

Pattern: clients use an event monitor that periodically polls the service, compares its state with the previous one and notifies the clients about changes

Notification

1. Event Monitor
2. Observer
3. Publish/Subscribe
4. Messaging Bridge
5. Half Synch-Half Asynch

Event Monitor



A client instructs the Event Monitor to periodically poll the state of a remote service. The Event Monitor only notifies the client if a change has occurred between different snapshots.

Event Monitor

- Necessary only if a service (or a component) does not provide a publish/subscribe interface
- To minimize the amount of information to be transferred, snapshots should be tailored to what events the client is interested in detecting. This may be a problem if the service interface cannot be changed.
- One event monitor should be shared among multiple clients by using the observer pattern
- Clients should be able to control polling frequency to reduce the load on the monitored service
- Warning: this solution does not guarantee that all changes will be detected, but only those which occur at most as often as the sampling rate of the event monitor

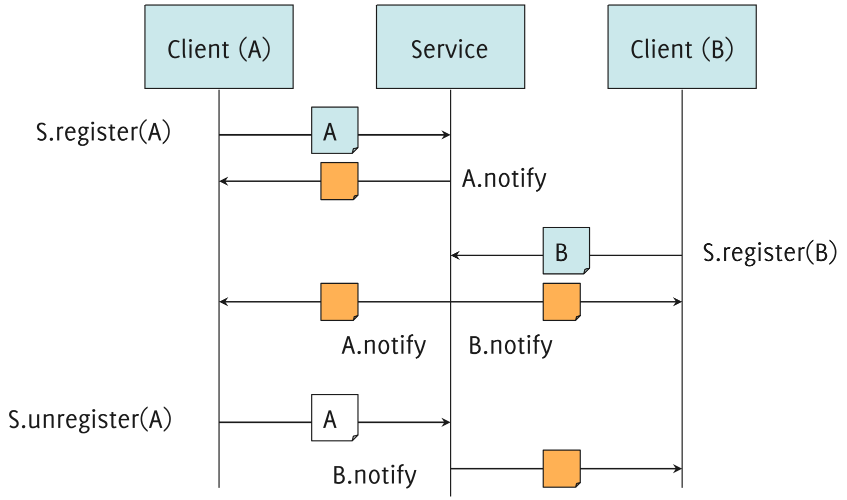
Observer

Goal: ensure a set of clients gets informed about all state changes of a service as soon as they occur

$\boxed{\text{detect changes and generate events at the service}}$

Pattern: clients register themselves with the service, which will inform them whenever a change occurs

Observer



Clients register a callback/notification endpoint with the service.
This is used by the service to notify all clients about state changes.
Clients should also unregister themselves when no longer interested.

Observer

- Assuming that a service can implement it, this pattern improves on the Event Monitor pattern:
  - State changes are not downsampled by clients
  - State changes are propagated as soon as they occur
  - If no change occurs, useless polling by clients is spared
- Clients could share the same endpoint with several observed services. Notifications should identify the service from which the event originates.
- To avoid unnecessary polling by the client, notification messages should also include information about the new state and not just that a state change has occurred.
- Warning: the set of registered clients becomes part of the state of the service and may have to be made persistent to survive service failures.
- A client which stops listening without unregistering should not affect the other registered clients

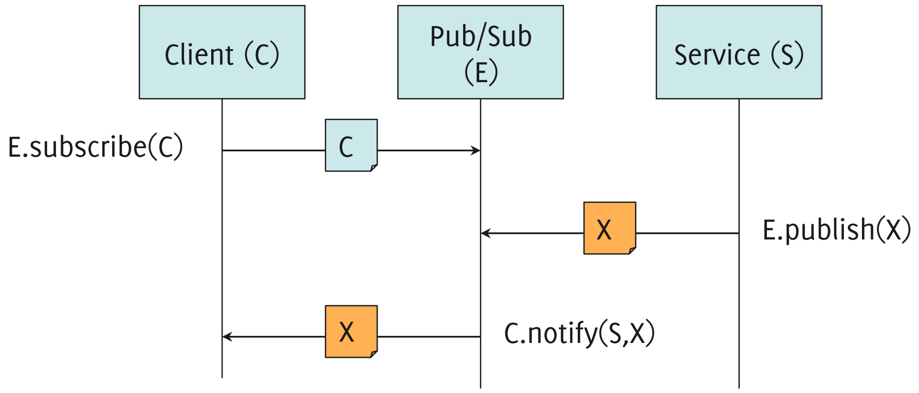
Publish/Subscribe

Goal: decouple clients from services generating events

factor out event propagation and subscription management into a separate service

Pattern: clients register with the event service by subscribing to certain event types, which are published to the event service by a set of one or more services.

Publish/Subscribe



As opposed to the Observer, subscriptions are centrally managed for a set of services by the Pub/Sub Service.
As opposed to the Event Monitor, services actively push state changes into the Pub/Sub Service.

Publish/Subscribe

- The interaction between published and subscriber is similar to the Observer pattern. It is also more decoupled, as messages are routed based on their type.
- Warning: event types become part of the interface of a service, and it may be difficult to determine the impact of changes in event type definitions
- Like in the Observer pattern, subscriptions should be made persistent, as it is difficult for clients to realize that the pub/sub service has failed.
- Unlike with the Observer pattern, all events in the system go through a centralized component, which can become a performance bottleneck. This can be alleviated by partitioning the pub/sub service by event type.

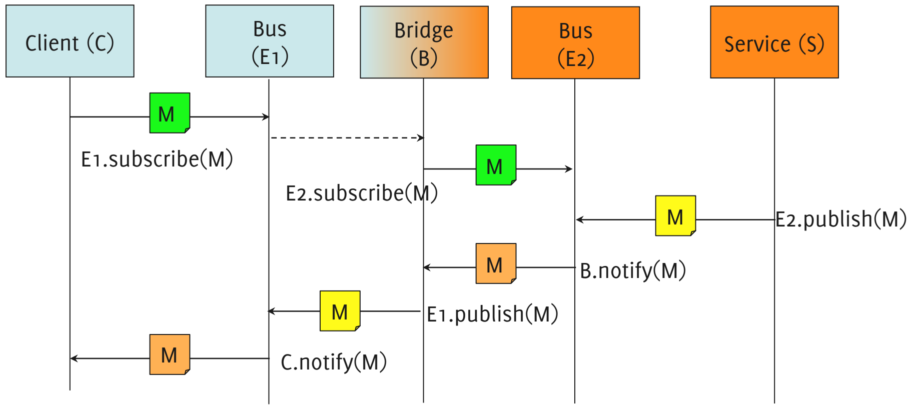
Messaging Bridge

Goal: connect multiple messaging systems

link multiple messaging systems to make messages exchanged on one also available on the others

Pattern: introduce a bridge between the messaging systems, the bridge forwards (and converts) all messages between the buses.

Messaging Bridge



The bridge (B) replays all subscriptions made by clients one bus (E1) onto the other (E2) so that it can inform the client (C) of one bus of messages published on the other

Messaging Bridge

- Enable the integration of existing multiple messaging systems so that applications do not have to use multiple messaging systems to communicate
- Often it is not possible to use a single messaging system (or bus) to integrate all applications:
  - Incompatible messaging middleware (JMS vs. MQ)
  - External bus(es) vs. Internal bus
  - One bus does not scale to handle all messages
- Although messaging systems may implement the same standard API, they are rarely interoperable so they cannot be directly connected.
- Messaging bridges are available (or can be implemented) so that messages can be exchanged between different buses. They act as a message consumer and producer at the same time and may have to perform message format translation.

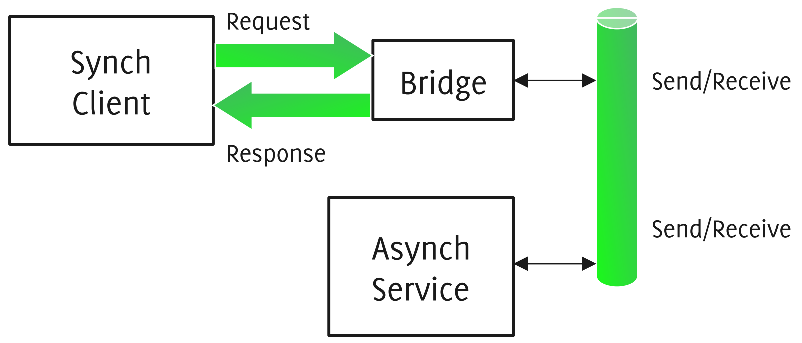
Half-Synch/Half-Asynch

Goal: interconnect synchronous and asynchronous components

Add a layer hiding asynchronous interactions behind a synchronous interface

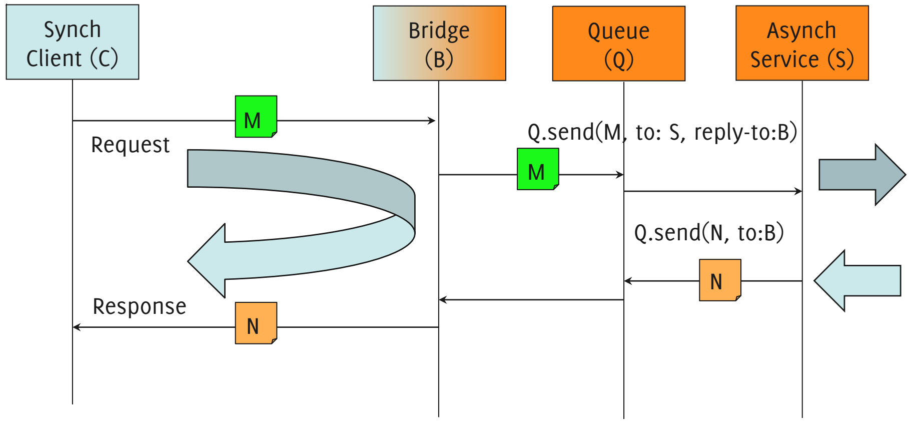
Pattern: The intermediate layer buffers synchronous requests using queues and maps them to the low-level asynchronous events.

Half-Synch/Half-Asynch



The bridge adapts from the synchronous **call** connector towards the client towards the asynchronous **bus** connector towards the service

Half-Synch/Half-Asynch



The bridge (B) queues the request M, blocks until it receives a response notification N from the queue Q and sends it back to the synchronous client C

Half-Synch/Half-Asynch

- Asynchronous events are more difficult to program with compared to synchronous function/method calls
- Asynchronous interfaces can give better performance:
  - Closer to the hardware
  - More efficient implementation
- Advantages:
  - Simplify access to low-level asynchronous interfaces (no need to deal with buffers, interrupts, and concurrency)
  - Separation of Concerns (each layer can use the most efficient implementation)
  - Centralized Interactions (all communication happens through the queues)
- Disadvantages:
  - Potential performance loss due to layer crossing (data copy, context switching, synchronization)