

**Name:**

Event-based, Implicit invocation

**Problem:**

As we build scalable systems, sometimes we want to distinguish or differentiate between the intent to take a certain action from the actual implementation of that action. How do you support such a system with a scalable solution?

**Context:**

This problem occurs in several domains, and typically the “intent” to perform an action is represented by an event. Many systems have independent reactive objects/processes/tasks/components that asynchronously post events to a medium that dispatches these events to interested components.

Consider a set of entities that process asynchronous events. For example, GUI controls on the screen that process mouse or keyboard events, or a server that monitors environment, stock prices, news updates. These entities, components in a larger system, can post events to a medium/environment to notify other interested components of what has occurred. Components that post an event do not know if any other component is interested, neither do they know in what order the events from different posters will be received by the interested parties. At the same time, the solution should scale well with possible large number of events, announcers, and listeners, i.e. the dispatch latency should not be “unreasonable”.

The pattern refers to the following terms that we define as follows:

- Component – a logically/physically independent entity that can have a dual role: (1) announcer -- it posts events to a medium; (2) listener -- it registers with a medium to receive certain events of interest. Components may be a part of a single software systems (e.g. GUI), or may be distributed (e.g. Internet Services)
- Event – a notification or a signal that can be posted to a medium. The event must contain an identifier for “event type” or “event source,” since either of these parameters enables the interested parties (components) to register to receive the event. The event may contain data. An event often contains a timestamp.

- Medium – a mechanism that performs two key functions: (1) it enables the components to register to receive the events of their choice, (2) when a component posts an event, the medium dispatches the event to the registered parties.

### ***Forces:***

- 
- A simple solution is requires a centralized medium implementation, but may suffer from high event dispatch and listener registration time. Load balancing is significantly simpler with a centralized medium.
- To ensure a highly scalable solution that does not force artificial serialization of the registering or dispatch of events, a distributed implementation of the medium (manager) might be required. A distributed approach, however, makes it more difficult to guarantee fairness and livelock freedom (in a sense that
  - an event must always be delivered to all registered listeners).
  - . The load balancing is complex with a distributed medium implementation.

### ***Solution:***

The Medium that performs listener registration and event dispatch essentially determines the performance and the effectiveness of the implementation of this pattern. The Medium includes the manager and the communication fabric. The manager maintains a list of registered components, receives event postings and performs dispatch. The communication fabric delivers the dispatched messages, and the implementation of a scalable manager depends heavily on the communication fabric.

There are several variants of components that can be considered. The components can all run in a single execution thread, or can run as independent threads. The key differentiating implementation factor, which strongly affects the parallelism available in the system, is the way dispatch is implemented. Some possible implementation are blocking procedure calls, non-blocking procedure calls, and sending a message. Assuming a single thread of execution, the event dispatch is a blocking procedure call. Alternatively, it is possible for the manager to spawn off a new thread with the registered listener, so that the listener can process the event (similar to a non-blocking procedure call). Finally, a manager may simply send a (network or inter-process) message to the listener component (which in practice is a local or remote

thread/process) to notify it of the event. If listeners can run as independent processes or threads, then the issue of load balancing and thread migration may be critical to high overall performance on a parallel architecture.

Manager implementation must fit the communication fabric to improve scalability. Although logically, the pattern assumes a single, centralized medium, and thus a single manager, its operation can be distributed in a number of ways. (1) Instead of a single manager, each component may be responsible of maintaining its own list of registered listeners and dispatching the event notifications. (2) The system might have several managers that are responsible for different types of events or for picking and dispatching events from certain components. (3) Managers can be distributed “geographically” and hierarchically on the network to essentially perform multi-casting of event notifications.

Consider a traditional case of a GUI. The communication fabric is memory. The graphical controls post events via procedure calls to an event manager (queue). The manager simply executes in the same or independent thread as the listeners, and dispatches the events as they become available via blocking procedure calls. This can also be implemented without a centralized manager, but the result is essentially the same. The main impediment to a scalable implementation is the centralized manager, but the approach is quite acceptable given that the event rate in the GUI is determined by human interactions.

Consider a distributed system, such as a server on the Internet, where the clients must register to receive events of interest. The communication fabric consists of network links, routers, load balancers, *etc.* Efficient implementation rests on the medium’s ability to multi-cast the notifications to the clients. In a closed system, such as a multi-core processor, one can imagine a distribution network that minimizes needless replication of events and data, and thus reduces on-chip network congestion.

Consider another example, Instant Messaging (e.g. MSN, Gtalk, YM, ...). When people talk to each other or in group, a person posts (sends) a message to servers (the managers) then other people will be notified by the servers and the servers will dispatch that message to them if they “registered” with the servers that they are chatting with that person. This is the case that the managers should be distributed “geographically” and hierarchically.

Although, the “event-based, implicit invocation” pattern in its purest form assumes that the announcer does not know whether any listener is interested in the event, in practice

a framework offering this pattern may offer “explicit invocation” that enables the announcer to notify another component directly. Of course, the other component should be free to ignore the message.

### ***Invariant:***

Pre-condition: An event is generated by an announcer. A listener for that type of event may or may not be registered.

Invariant: Announcers of events do not know which listeners will be affected by those events.

Post-condition: If a listener is registered, the system will invoke the callback registered.

### ***Examples:***

GUIs

Control, listening for interrupts, interfacing with the outside physical world.

Clients registering with an over the network server to receive notifications for certain events (stock price updates).

Instant messaging (MSN, GTalk, or Yahoo messenger).

### ***Known Uses:***

Windows or X11 GUIs. Operating systems send signal to processes that often ignore or handle only a subset of signals.

Google Android Mobile Phone Framework’s *Intents* allow many different *IntentHandlers* to be called by the system to satisfy intents from different applications, so that the system can be customized in many different ways.

Registering callback in the network stack.

### ***Related Patterns:***

The following PLPP patterns are related to Event-based, Implicit Invocation

- Data-flow/Pipeline – Event-based pattern can be used to orchestrate a data-flow computation with static or dynamic actor graph topology.

## ***Authors***

Yury Markovsky, Dai Bui, Hugo Andrade, Jimmy Su