**Broadcast Trigger**

**Author**

Dietmar Schütz  
Siemens AG, CT SE 2  
Otto-Hahn-Ring 6  
81739 München  
Germany  
EMail: dietmar.schuetz@siemens.com  
Phone: +49 (89) 636-57380

**Summary**

The "Broadcast Trigger" design pattern focuses on the co-ordination of distributed autonomous components, which are co-operating in order to accomplish a common task, for example a robot assembling a mobile phone. The pattern enables a central coordinator to control the distributed consecutive actions using a time-optimised interaction scheme. Actions are prepared from "aside the critical path" during auxiliary process time, by sending them to the responsible component, guarded by a precondition.

**Example**

Consider an assembly robot in a production line for mobile phones. Such a robot consists of different intelligent subsystems, interconnected by a field bus like CAN¹. Major subsystems of the robot are: a conveyor belt to transport the partly assembled phones, a feeder unit for the parts to be mounted, and a flexible arm, carrying a head with various tools such as grippers and optical inspection systems. The assembly process is specified by means of a machine program, which defines a lot of single steps for each part, to be carried out by the subsystems in a mixed sequential and concurrent order.

The software operating the robot is primarily an interpreter for machine programs, acting as a central coordinator. The program steps are performed strictly maintaining the appropriate order, by issuing the corresponding commands and parameters to the participating subsystems as soon as all mandatory preceding actions have been completed. In complex scenarios, the number of steps necessary to handle one single part can exceed 30, most

---

¹ Controller Area Network, see [CAN]
of these steps located on the critical path. Obviously, latencies in communication and response across the bus sum up to significant delays, decreasing the performance of the robot. Furthermore, the repetitive actions of the robot develop to a rhythmic noise the operators get used to. Therefore, jitter in response causes variations in the rhythm, which are heard easily, generating the (true) impression that the robot is wasting valuable time instead of being productive.

**Context**

Reactive (typically embedded) systems, where independent (distributed) active subsystems are controlled and co-ordinated by a centralised application, and quick responses are crucial.

**Problem**

After one of the working subsystems has finished its current task, a completion message is sent to the central coordinator. The coordinator determines the successive actions and the designated subsystems, and initiates these actions by sending the corresponding commands to the subsystems. Obviously, a subsequent actions that could be performed directly after its predecessor, is delayed by two communication steps to and from the coordinator, plus some intermediate computation. Instead of wasting time on the critical path, how can the time span between to actions be minimized?

The following **forces** influence the solution:

- The overall flow of control and sequence of commands is subject to change. Changes usually happen at start up, but may also be necessary during run-time.

- Since the program uses the subsystems to achieve the required result, the program is correlated to the configuration of the whole system as well as its subsystems.

- For the sake of de-coupling, the subsystems must be autonomous, not depending on each other.

- Starting subsequent actions ahead of time can cause severe damage if the preceding actions fail to complete successfully.

- Latencies of the communication network or within the coordinator component may lead to unpredictable response times.

- Using a real-time operating system leads to predictable response times, but real-time is not necessarily real fast, and things still take place right on the critical path.

**Solution**

Extend the commands with a trigger: a precondition, defined by means of a compound system state that reflects the status of all subsystems. Send commands to subsystems in advance (during auxiliary process time) in order to prepare the subsystems for the oncoming execution. Have the subsystems report the completion of their tasks by broadcasting the resulting state across the network. Let all subsystems monitor the system state, and
start prepared commands on their own (without detouring via the coordinator) as soon as the precondition applies.

**Structure**

The Coordinator component is the "central" instance that manages the correct order of execution of the overall program steps. To this end, the coordinator sends 'prepare'-messages (commands, guarded by a precondition) to the respective subsystems. Preparation is usually done during auxiliary process time, in time before the preceding commands. From outside the system, the coordinator often acts as a facade for the various subsystems.

A Subsystem provides all capabilities necessary to execute commands. The interface of a subsystem offers a method to prepare a command to be executed in association with a precondition that must apply in order to initiate the execution. In addition, the interface supports cancellation of previously prepared command in order to handle error scenarios.

Class Coordinator

**Responsibility**

- supervision and control of commands and active subsystems
- monitor the system state
- issue "prepare"-commands in accordance with the program

**Collaborators**

- Subsystem
- Command
- SystemState

Class Subsystem

**Responsibility**

- queue commands
- monitor the system state
- execute commands autonomously when precondition applies

**Collaborators**

- Coordinator
- Command
- SystemState

Command is a common abstraction of a function request. Every concrete instance of command contains all parameters that shape its execution, ready to be carried out by the designated subsystem.

The SystemState is a conglomeration of the processing state of all subsystems. It provides an interface for subsystems to update/modify the state in accordance with their own status and progress. The SystemState also is the base type for preconditions, and supports an equal-method that allows comparing preconditions with the current state.

Class Command

**Responsibility**

- Defines a uniform interface to specify, create and perform actions/tasks on a subsystem
- Encapsulates a function request

**Collaborators**

- Coordinator
- Subsystem

Class SystemState

**Responsibility**

- Compound processing state of all subsystems
- Interface for update by a single subsystem
- Serves as type for preconditions

**Collaborators**

- Coordinator
- Subsystem
The following class diagram shows the relationships between these participants.

![Class Diagram](image)

**Dynamics**

The scenario below depicts the preparation of a command and its triggered execution. While some subsystems are working on their tasks, the Coordinator issues a command to ComponentA, where the command is queued since its precondition is not valid yet. As soon as ComponentB finishes its task, the resulting state is broadcasted across the bus. All components update their system state image and check their queues for commands with a precondition matching the new system state. ComponentA discovers the command prepared up front and starts its execution.
The next scenario shows the behaviour of the system in case if the Coordinator misses to prepare a command in time. ComponentB finishes its task, and the resulting state is broadcasted across the bus. All components update their system state image and check their queues, but do not find a precondition matching the new system state and therefore do not start any actions. Later on, the coordinator issues a new command for ComponentA, which discovers that the precondition matches the current state and starts the execution of the command immediately.

Given this scenario, the execution of the command by ComponentA is delayed, but at least it is not lost.

**Implementation** To implement the pattern, carry out the steps described below.

1. Define the representation for the SystemState. The SystemState must meet the following requirements:
   - For each command, the system state must change upon its execution.
   - Updating the SystemState must be possible for a single Subsystem, without any knowledge about or interaction with other subsystems or the coordinator.

A plausible approach to define the SystemState is a composition of the local states of each subsystem. This obviously allows an independent update on the system state, since only the local state of a subsystem needs to be replaced with the new one. This also reduces the focus of the first requirement from the whole system to the local context each single subsystem; therefore, the requirement can be achieved easily.
The SystemState of the assembly robot is stored as an array of integers, indexed by subsystem Id's, containing the local state of each subsystem.

```cpp
const int MAX_SubSystem = 10;
enum {eCoordinator = 0,
    eSubSysConveyor = 1,
    eSubSysHead = 2,
    ...,
    eNotUsed3 = 9};
typedef int tSubsystemState;
class SystemState {
    tSystemState[MAX_SubSystem] subSystemState;
    update( int iSubSysId, tSubsystemState newSubSysState) {
        subSystemState[iSubSysId] = newSubSysState;
    }
}
```

With respect to further extensions, the array is oversized, providing some spare indices that are set to a constant default state. This decision allows adding new subsystems without changing the software of existing ones.

2 Design and implement the Command class. The design and usage of the Command class is an instantiation of the Command Processor pattern in [POSA96] and its mental origin, the Command pattern in [Gof94]. Command is an abstract class, providing a method to execute itself.

Since Commands reside in different address spaces and are sent across the bus as a parameter of the prepare method, Command provides a virtual method to serialize itself, and a class method to create a new Command from a given serialized representation.

```cpp
class Command {
    static &Command create(string serialRep);
    virtual int execute() = 0;
}
```

3 Define and implement the Subsystem class. Each subsystem maintains a list of commands and associated preconditions, storing commands that have been prepared, but not executed yet. The prepare method of the Subsystem interface is used to add new commands to this list. A counterpart method allows deleting prepared commands, which is necessary whenever some previous command fail to complete successfully.

Additionally, subsystems are monitoring the system state by keeping track of change notifications on the bus. Therefore, the subsystem interface must contain a callback method that is invoked if the corresponding message arrives on the bus. Internally, this method updates the local representation of the system state and checks the command list for prepared commands.
Prepared commands are stored in map that is indexed by the precondition.

class SubSystem {
  Systemstate curSystemState;
  map<Systemstate, Command> preparedCmds;
  updateSysState(int iSubSysId, tSubsystemState newSubSysState) {
    Command *cmd;
    systemstate.update(iSubSysId, newSubSysState)
    while ( cmd = preparedCmds.extract(curSystemState)
      if ( cmd.execute( resultingState ) )
        notifySuccess( mySubSysId(), resultingState )
      else
        notifyFail( mySubSysId(), resultingState )
    }
  }
  int prepare( Command cmd; SystemState precond)
    if ( precond == curSystemState) {
      // execute command immediately
      if ( cmd.execute( resultingState ) )
        notifySuccess( mySubSysId(), resultingState )
      else
        notifyFail( mySubSysId(), resultingState )
    } else
      preparedCmds.add(precond, cmd)
  }
  void cancel( Command cmd; SystemState precond) {
    preparedCmds.delete(precond, cmd);
  }
}

4 Derive concrete commands and concrete subsystems. Sub-classing is according to the rules defined by the base class. Every ConcreteCommand is related to a ConcreteSubsystem that provides support to execute the command.

5 Determine and implement the preparation policy. The coordinator must implement the code for sending preparation messages in accordance with the machine program. The most important step is to balance the preparation lead-time. A short time span between preparing a command and its precondition coming true increases the probability of delays caused by late preparation. On the other hand, preparing commands long in advance may increase the risk of a premature execution, as explained below.

Typically, some subsystems iterate through the same sequence of states many times, thereby performing the same commands, but with different parameters, for example when placing two identical parts at two different positions. The preparation of such similar but not the same actions gives room for a dangerous scenario: assuming the SystemState is a composition of the subsystem states (as implemented by the example in step 1), the system reaches the “same” state repeatedly. Hence, a long lead-time increases the risk that different commands are prepared under the same precondition. This would cause a premature start of some commands, with unpredictable consequences.

The preparation policy is also influenced if the overall task contains alternative execution sequences where the decision depends on the result of
a previous step. That being the case, the simple approach is to delay the
preparation until the necessary input is available, an only prepare actions
from the path to follow. This policy increases the risk of a late preparation.
If this possible delay cannot be tolerated, all alternatives need to be
prepared. As soon as the decision is determined, the superfluous
preparations must be deleted; this can for example be done by a meta-
command\(^2\) that is embedded into the opposite alternative.

Within the assembly robot, various parts are handled subsequently using
the same pick and place sequence, but with different source and target
positions. And since the system state is composed from the states of the
subsystems, the “same” system state is entered repeatedly. As a result, the
preparation policy is quite restrictive: commands are prepared when the
coordinator detects that a predecessor of a command has started its
execution. The strategy makes direct usage of the fact that system states
are different before and after executing a command, hence definitely preventing
premature start. This leaves a noticeable risk of delays, especially after
extremely short actions, but a more intense preparation has been agreed on
to be too confusing.

The assembly process also incorporates alternative execution paths. For
example, every part to be mounted is inspected with a camera in advance,
and only “good” parts are handled as usual. If a part violates its
specification, it is dropped into a garbage box. In these cases, a defensive
preparation policy is chosen, since it is easier to implement and does not
enforce to introduce the new category of meta-commands.

\(^6\) Implement the coordinator. Since the preparation behaviour is a simple add-
on to a centralized control, the implementation of the coordinator first
focuses on managing the correct sequence of actions, without paying respect
to preparation. Given the control structure, for example by means of a finite
state machine, the coordinator uses a call-back policy to react to events
indicating state changes.

As soon as this mode of operation is established and stable, the transition to
preparation can begin. This is done by replacing the code issuing commands
to the subsystems with the corresponding preparation request. Furthermore,
the condition used to trigger preparation is attached to the monitored begin
of subsequent actions.

In the first implementation of the assembly robot coordinator, all
command calls to the subsystems are encapsulated by a common interface.
This fact supports the transition to preparation behaviour: within the generic
implementation, issuing of commands is substituted by sending preparation
messages, and the call-back interface is fed by “action started” events

\(^2\) This command does not represent a physical action of the subsystem, but a
change in control flow.
instead of completion events. This automatically couples the preparation to the first action that will generate the precondition.

Special attention is necessary for situations where command execution fails. Given that, the follow-up commands prepared by now must be cancelled in order to prevent executing them under wrong circumstances. After cancellation, the coordinator must recover from that situation, typically by starting and preparing an alternative command sequence. An additional error scenario to be covered is failure during preparation, caused for example by an overflow in the data structures storing the prepared commands.

7 **Bus Optimisation.** Many field busses used in an environment where this pattern applies offer mechanisms for prioritising messages. Even with an average bus load clearly below the limit, careful tuning is necessary to guarantee optimal performance for the different communication scenarios.

In general, preparation commands are associated with the lower priorities. They are to be sent long in advance, and there is enough time to submit them during quiet phases. On the other hand, completion messages use higher priorities, since they need to be transmitted as soon as possible. An even higher priority must be reserved for “emergency” preparation, which occurs if the coordinator detects that a precondition have become true without the subsequent commands prepared yet.

The CAN-bus used within the assembly robot offers bit arbitration, applied on the message id: speaking only in terms of dominant and recessive values\(^3\) for a single bit, concurrent transmission is aborted by a participating device as soon as it detects that his recessive bit value is overridden by a dominant one from somebody else. Emergency preparation messages from the coordinator are indicated by a dominant value on the first bit. The second bit distinguishes completion messages (dominant) from normal preparation (recessive). The remaining bits uniquely identify the sender, using the arbitration hierarchy to favour subsystems with high traffic load.

The software of the assembly robot is implemented on top of a common non-real-time operating system, such as Windows NT™ or Linux, using a standard development environment. The integration of controller and operation interface introduces no additional pitfalls. There is no need for more expensive licenses and tools, and engineers do not need specific knowledge.

---

\(^3\) Depending on characteristics of the physical layer. For example, within an optical system light is dominant over darkness. The logical values associated with dominant and recessive is not standardized, although most implementations follow the intuitive approach of a dominant “high” or “one” value.
The robot provides enough flexibility to run different programs, even with runtime changes. Nonetheless, the machine is notably fast under normal conditions.

**Variants**

*No central coordinator.* The main role of the coordinator is to provide flexibility in command execution during run-time. If this flexibility is not mandatory, the coordinator can be omitted, including the whole preparation behaviour. Instead, every subsystem hosts a local state machine that implements the subsystem-specific part of the overall flow of control, still based on the abstract idea of system states. Given this model of cooperation, using a broadcast completion message still decouples the subsystem from each other and provides minimal response times.

*Enrolled Petri network.* An alternative approach to define the SystemState derives from the observation that a machine program can be described by a directed graph, consisting of synchronisation spots as nodes, and commands as edges. Such a structure is known as an enrolled Petri-network\(^4\); it contains no cycles or loops.

Based on this, the precondition of a command can be defined by a unique Id, representing one node from the graph. To monitor progress when approaching this precondition, it is extended by a counter that is initialised with the number of edges leading into that node. Every successful completion of a command broadcasts the Id of its target node, and all monitoring components decrease their corresponding counter instance by one. A node is “reached” and the precondition becomes true as soon as the associated edge counter drops down to zero.

This approach allows a nearly infinite preparation of commands, with no risk of premature execution, since every synchronization node has its own unique Id. Hence, there is almost no need to think about a preparation policy (see implementation step 5). One significant limitation may be the availability of memory to store and keep track of the preconditions and commands on the subsystems. In addition, there is a (minimal) risk that uniqueness of node Id’s cannot be guaranteed during overly long continuous operation.

**Known Use**

The new generation of Siemens placement machines (branded Siplace) uses this pattern in the implementation of the machine controller, based on a non-real-time Microsoft-Windows™ operating system.

By now, I haven’t got notice of other uses, but I’m sure they exist. So if you run into one, please let me know.

---

\(^4\) Whereas a usual Petri-network contains cycles and is generated from the usually cyclic base Petri-network
Consequences

The Broadcast Trigger pattern provides the benefits depicted below:

- **Minimal response time.** The time span between two subsequent actions of distributed subsystems is minimal, since there is only one message in-between.

- **Predictable response time.** The prediction of response time characteristics (average, maximum, jitter) is much easier, since the number of participating components is minimal.

- **Decoupling of subsystems.** The subsystems are completely decoupled from the overall task of the system. The subsystems don’t need to know specific details from each other. Hence, the machine is still modular and provides the flexibility of replacing subsystems by other implementations.

- **Programmable.** The flow of control can be changed easily at start-up time, and also during runtime.

On the other hand, the pattern carries the following liabilities:

- **Complex preparation scenarios.** The elaboration and implementation of the preparation policy is a tedious job. Hence, this solution implies an increased risk of program errors.

- **Increased bus load.** In comparison to normal command issuing, the additional transmission of the system states during preparation increased the amount of data sent on the bus.

- **Complex error recovery.** If a subsystem fails to accomplish its task, some kind of error recovery has to take place. For example, all prepared commands that depend on the failed one must be revoked.

- **Hard to debug.** Since preparation and execution of a command are separated by a (nearly unpredictable) time span, debugging the application is much more difficult.

See Also

The implementation of the subsystems will benefit from adapting the Command-Processor pattern (see [POSA]) or the underlying Command pattern (see [GoF94]).

The coordinator is an example of inverted programming, a technique very common in user interface frameworks.

Credits

Thanks to my former colleagues at Siemens Dematic: Gerhard Haas, Robert Huber and Martin Prüfer. They developed the idea of preparing commands and received a patent [Patent] for it. They also supported me with clarifying this description.

Thanks to my shepherd Prashant Jain, he showed me a forest where I only saw trees.
### References

| [GoF94] | E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software; Addison-Wesley 1994 |