# Asynchronous Completion Token

## An Object Behavioral Pattern for Efficient Asynchronous Event Handling

Timothy H. Harrison, Douglas C. Schmidt, and Irfan Pyarali
harrison@cs.wustl.edu, schmidt@cs.wustl.edu, and irfan@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, Missouri, 63130

An earlier version of this paper appeared as a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

### Abstract

*Event-driven applications must respond to different types of events, ranging from user interface notifications to network messages. Delay-sensitive applications, such as network management systems, often perform operations asynchronously to avoid blocking the processing of other pending events. When these asynchronous operations complete, client applications may need more information than simply the completion event itself to perform the appropriate operation completion processing. This paper describes how such applications can benefit from the use of the Asynchronous Completion Token pattern, which allows applications to efficiently associate actions and state with the completion of asynchronous operations.*

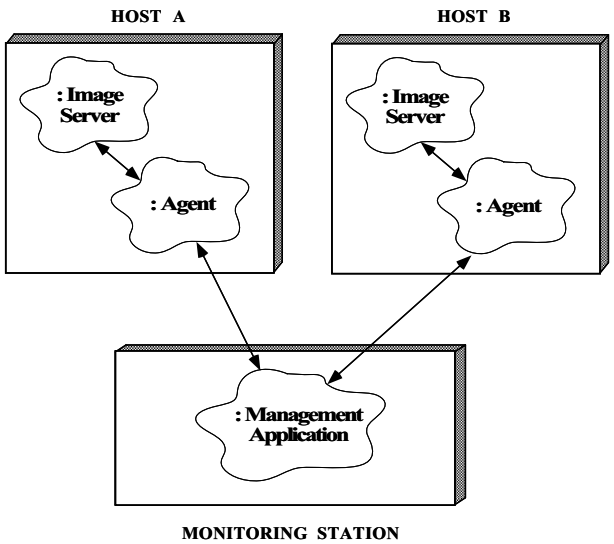


Figure 1: Participants in an EMIS Management System

# 1 Intent

The Asynchronous Completion Token design pattern efficiently associates the designated client actions and state with the completion of operations invoked asynchronously on services.

# 2 Also Known As

"Magic Cookie"

# 3 Example

To illustrate the Asynchronous Completion Token pattern, consider the structure of a network management application that monitors the performance and status of multiple components in a distributed Electronic Medical Imaging System (EMIS) [1]. Figure 1 shows a simplified view of the management application and several EMIS components. These components include *image servers*, which store and retrieve medical images [2], and *agents*, which interact with EMIS components on behalf of management applications.[1]

The performance and reliability of an EMIS is crucial to physicians, medical staff, and patients using the system. Therefore, it is important to monitor the state of EMIS components carefully. Agents address this need by propagating events from EMIS components, such as image servers, back to management applications residing on central monitoring stations. Administrators can use these management applications to monitor, visualize, and control [4] the overall status and performance of EMIS components in a distributed system.

Figure 2 shows a typical sequence of events exchanged between the management application, agents, and EMIS image server components. A management application uses an asynchronous operation to register **(1)** with one or more agents for periodic updates using the Publisher/Subscriber pattern [5]. For example, it might send a request to an agent asking to

[1] Other components in a distributed EMIS include modalities, clinical and diagnostic workstations that perform imaging processing and display, hierarchical storage management (HSM) systems, and patient record databases [3].

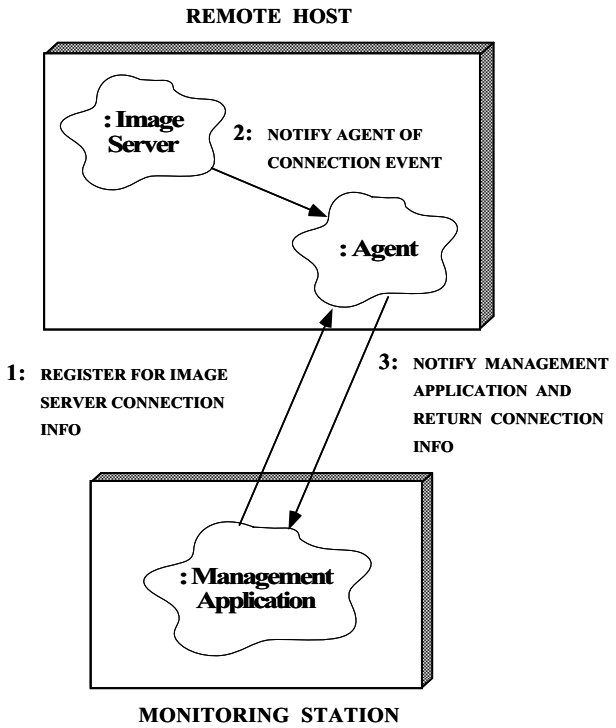


Figure 2: Event Registration and Callback

receive an event every time an image server on a particular host accepts a new network connection. When the agent on that host detects a new connection **(2)** to the image server, it sends a connection established *completion event* **(3)** to the management application. The management application can then **(4)** display the new connection graphically on its console.

Several application-specific actions must be performed by a management application whenever an agent sends it a completion event. For instance, the management application may need to update a graphical display or record the event in a database. A management application may have registered asynchronously for many types of events from many different agents. For each event completion notification, therefore, the application must determine the appropriate action, such as updating a graphical display or logging to a database, in an efficient and scalable manner.

One way a management application could match up asynchronous operation requests with their subsequent completion events would be to spawn a separate thread for each event registration operation it sends to an agent. Each operation would block synchronously, waiting until its agent replied. In this approach, the action and state information required to process agent replies could be stored implicitly in the context of each thread's run-time stack.

However, this synchronous multi-threaded design incurs the same performance, portability, and complexity drawbacks described in the Reactor pattern. As a result, developing management applications using a separate thread to wait for the completion of each operation can yield inefficient and overly complex solutions. Yet, the management application must associate service responses with client operation requests efficiently and scalably to ensure adequate quality of service for all its agents.

# 4 Context

An event-driven system where clients invoke asynchronous operations on services.

# 5 Problem

When a client invokes an operation asynchronously on a service, the service indicates its completion by sending a notification event back to the client. To distinguish between various types of notification events, a client must be able to associate actions and state with the completion of operations it invoked asynchronously. Addressing this demultiplexing problem efficiently and scalably requires the resolution of the following three forces:

**Minimal communication bandwidth:** The minimal information should be exchanged between client and service to reduce the amount of communication overhead. This is particularly important for distributed services accessed by clients over bandwidth-limited communication links.

**Minimal performance degradation:** A client should spend as little time as possible processing the data returned from the service in order to determine the action(s) and state required to process the completion of an asynchronous operation. In particular, searching a large table for each request is often unacceptable because it can degrade the performance of the client significantly.

**Application-defined context information:** It is difficult for a service to know what data its clients need to perform completion processing because it does not necessarily know the context in which the clients invoked asynchronous operations. Therefore, it should be the responsibility of the client, not the server, to determine what actions and state to associate with completion events.

# 6 Solution

Associate application-specific actions and state with events that indicate the completion of asynchronous operations. For every asynchronous operation that a *client* can invoke on a *service*, create an *asynchronous completion token* (ACT) and pass it along with the operation to the service. When the service replies to the application it must pass back the ACT that was sent to it originally, that is, the service must not change the value of the ACT. Therefore, the client can use the ACT to efficiently identify the state needed to perform the actions associated with processing the completion of the asynchronously invoked operation.

# 7 Structure

The following participants form the structure of the Asynchronous Completion Token pattern:

**Asynchronous Completion Token:**

- When a client invokes an operation asynchronously on a Service, it passes an Asynchronous Completion Token (ACT) to the Service. ACTs can be indices into tables or direct pointers to memory holding the actions and state necessary to handle the completion of the operation. To the Service, however, the ACT is simply an opaque object that it returns to the client unchanged when the asynchronous operation completes.

  In the EMIS example, an ACT is passed from a management application to an agent when the application registers to receive image server connection information. This ACT is associated with management application-specific completion processing actions and state, such as which GUI window to use to display a new connection notification.

**Service:**

- A Service provides some type of functionality to Clients. Services can hold a collection of ACTs; when an asynchronous operation completes the appropriate ACT will be returned to the associated Client.

  In the EMIS example, agents provide a service that propagates EMIS events to management applications asynchronously.

**Client:**

- A Client invokes operations on a Service asynchronously. To correctly process the completion of asynchronous operations, a client requires an ACT to be passed back from the Service in order to locate the associated actions and state necessary to process the operation completion.

  In the EMIS example, the management application is a Client to the agent Service.

Figure 3 illustrates the relationships between these participants.

# 8 Dynamics

The interactions between the participants of the ACT pattern can be divided into the following four phases:

**1. ACT creation:** Before invoking an asynchronous operation on the Service, the Client creates the ACT associated with operation.

**2. The asynchronous operation invocation:** When invoking the operation on the Service, the Client passes a reference to the ACT to the Service.

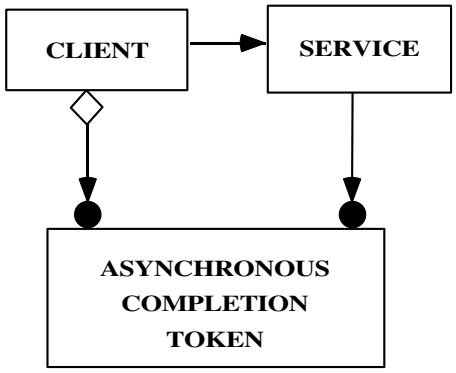


Figure 3: Structure of Participants in the Asynchronous Completion Token Pattern

**3. Client processing:** A Client can continue executing while the Service performs the operation because the operation is invoked asynchronously.

**4. Asynchronous operation completion:** When the asynchronous operation completes, the Service notifies the Client and returns the ACT. The Client uses the ACT to regain the needed state and apply the application-specific completion actions for that operation.

Figure 4 illustrates these phases.

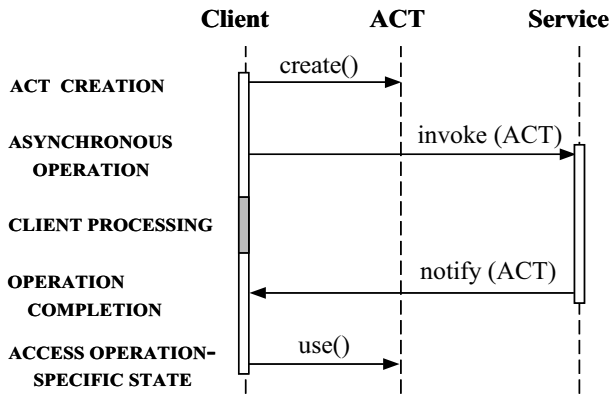


Figure 4: Participant Collaborations in the Asynchronous Completion Token Pattern

# 9 Implementation

The four steps involved in implementing the ACT pattern are described below.

**1. Define the ACT representation:** ACTs can be represented as `void *` pointers or as pointers to abstract base classes. This former approach is frequently chosen when implementing the ACT pattern with low-level programming languages, such as C or C++. When a Client initiates an operation on the Service, it creates the ACT and casts it to a `void *` pointer. It is important to consider potential platform differences that might represent `void *` pointers differently. In this case, the developers may choose to

use compiler and language constructs (*e.g.*, `defines` and `typedefs`) to ensure a uniform representation throughout the system.

When using a higher-level language or middleware, such as Java or CORBA, developers can represent ACTs as references to interfaces. A Service typically can not make assumptions about the use of the ACTs. Therefore, ACT interfaces will be (mostly) empty, for instance, a CORBA `Object` interface. Upon receiving an ACT, a client can use dynamic casting to narrow the ACT to a type that's meaningful to it.

**2. Determine how to pass the ACT from Client to Service:** This step is typically straightforward. Clients can pass ACTs as explicit or implicit parameters to asynchronous operation invocations. In section 10 we show an example of how ACTs can be parameters to Service methods.

**3. Determine a mechanism for hold the ACT at the Service:** Depending on the Service, this step can be simple or complex. If the Service operates synchronously, then the ACT can be held on the run-time stack while the Service is processing the operation. For instance, if the Service is running in a separate thread or process from the Client, it can perform its operations synchronously, while still providing an asynchronous programming model to the Client [6].

Services can *internally* use asynchronous services and may need to handle multiple requests simultaneously. For instance, consider a management application that performs requests on an agent, which in turn performs requests on a timer mechanism. This scenario can be thought of as a Client (the management application) using a chain of Services. All members in the chain (except the two ends) are both Clients and Services because they both receive and initiate requests.

If each member of the chain uses the ACT pattern as Service, there are several choices when performing requests as Clients. If a member does not need to add state to the operation, it can simply pass along the original ACT received from the previous Client. However, when additional state needs to be associated with the operation, that state must contain the original ACT and a new ACT must be created. This is because the Service can not normally make any assumptions about the values of received ACTs. If a Service was guaranteed that ACT values were unique, then it could use them as indexes into a data structure mapping ACT to operation state. However, if uniqueness can not be assumed, the original ACT can not reused to reference new state.

**4. Determine a demultiplexing mechanism to dispatch the ACT on the Client:** In some applications, when a Client initiates an asynchronous operation on a Service, the ACT is returned just once, *i.e.*, with the response. In other applications, however, ACTs from a single request may be returned multiple times. For instance, if a multiple responses are tagged with the same ACT, the receiving Client can use the same ACT to demultiplex and dispatch many replies. The EMIS management application uses the multiple response model in order to implement the Publisher/Subscriber pattern features described in Section 3.

In both the single and multiple response models, there are several options to *how* ACTs are returned to Clients when asynchronous operations complete. Thus, you must decide the demultiplexing mechanism used to notify clients when asynchronous operations complete. There are several common approaches:

• **Callbacks:** In the callback approach, Clients specify functions or class methods that are called by the Service when an operation completes [7]. Depending on the Service, callback functions can be specified once or on a per-request basis. In the callback approach, the ACT is returned as a parameter to the callback function. Section 10 shows an example of the callback approach.

• **Queued completion notifications:** This approach queues up completion notifications, which can be retrieved at the Client's discretion. Win32 I/O completion ports use this approach. When Win32 handles[2] are created, they can be associated with completion ports through `CreateIoCompletionPort`. Completion ports provide a location for completion notifications to be queued by kernel-level Services and dequeued by Clients. When Clients initiate asynchronous reads and writes via `ReadFile` and `WriteFile`, they specify `OVERLAPPED` structures that will be queued at a completion port when the operations complete. Clients dequeue completion notifications (including the `OVERLAPPED` structures) via the `GetQueuedCompletionStatus` function.

**Asynchronous callbacks:** A variation on the callback approach is used by implementations of POSIX 4.0 Asynchronous I/O [8]. Clients can specify that completion notifications for asynchronous I/O operations be returned via UNIX signals. This approach is similar to the callback approach, *i.e.*, a registered signal handler is called. However, it differs because Clients need not explicitly wait for notifications, *e.g.*, they need not block in an event loop.

Consider the chain of Services example described earlier. In addition to deciding how ACTs are returned to Clients, a chain of Services must decide which Service calls back the client. If no Service in the Chain created new ACTs to associate additional state with the operation, then the last Service in the chain can notify the Client. This would optimize the process because, in this case, "unwinding" the chain of Services is unnecessary. In the EMIS example, agents associate additional state with the timer operations, so "unwinding" is required.

Regardless of how the ACTs are returned to the Clients, once they are returned, the job of the Service is done. Clients are responsible for handling the completed operation and freeing any resources associated with the ACT.

[2]For Win32 overlapped I/O, handles are used to identify network connection endpoints or open files. Win32 handles are similar to UNIX file descriptors.

## 10 Example Resolved

The following discussion illustrates how to apply the Asynchronous Completion Token pattern to develop the management application portion of the distributed Electronic Medical Imaging System (EMIS) described in Section 3. To make the example concrete, consider a typical scenario where an EMIS administrator uses a management application to log the connections made to a particular image server. The management application must first invoke an asynchronous operation that registers with the image server's agent to be notified of connection events. When connection established *completion events* are sent by the agent, the management application must know to log the data and update its GUI accordingly.

The Asynchronous Completion Token pattern addresses this need by allowing the management application to pass an opaque value as an ACT. For instance, when registering with the agent, the management application can pass a reference to a State object as the ACT. The State object contains references to a logging object and the appropriate GUI window that will be updated when connection events arrive. The management application can use the State object referenced by the ACT to update the correct user interface and record the event with the appropriate logging object.

Figure 5 shows the order of events in such a scenario. Before the management application registers with the agent,

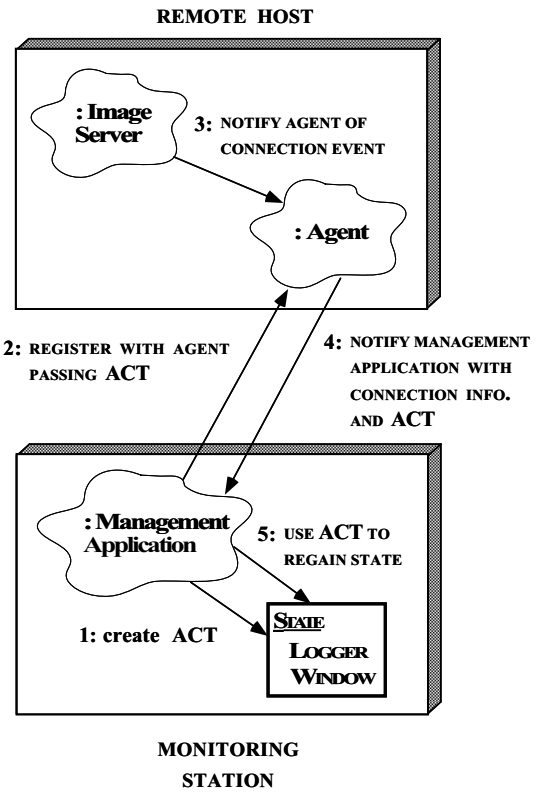


Figure 5: Event Registration and Callback with ACTs

it creates the various `State` objects needed to handle its completion events. Next, the management application passes a reference to the State object as an ACT. When a connection is created at the image server, the agent is notified and subsequently calls back the management application. The management application receives the connection information and the ACT, which it uses to guide subsequent the event-specific completion processing actions.

The C++ code below uses EMIS management applications and agents to illustrate the use of ACTs with asynchronous I/O operations. The following code defines an implementation of a `Management Application` class that handles asynchronous EMIS events received from agents.

```
// Use a generic C++ pointer.
typedef void *ACT;

class EMIS_Event_Handler
  : public Receiver
    // Defines the pure virtual recv_event() method.
{
public:
  // References to States will be passed
  // as ACTs when invoking asynchronous operations.
  class State
  {
  public:
    State (Window *w,
           Logger *l)
      : window_ (w),
        logger_ (l)
    {}

  private:
    Window *window_; // Used to display state.
    Logger *logger_; // Used to log state.
    // ...
  };

  // Called back by agents EMIS when events occur.
  virtual void recv_event (const Event& event,
                           ACT act)
  {
    // Turn the ACT into the needed state.
    State *state = reinterpret_cast <State *> (act);

    // Update a graphical window.
    state->window_->update (event);

    // Log the event.
    state->logger_->record (event);

    // ...
  }
};
```

The following code defines the `Agent` interface that can be invoked by clients to register for EMIS completion events.

```
class Agent
{
  // The types of events that applications
  // can register for.
  enum Event_Type
  {
    NEW_CONNECTIONS,
    IMAGE_TRANSFERS
    // ...
  };

  // Register for <receiver> to get called
  // back when the <type> of EMIS events occur.
```

```
  // The <act> is passed back to <receiver>.
  void register (Receiver *receiver,
                 Event_Type type,
                 ACT act);
  // ...
};
```

The following code shows how a management application invokes operations on an `Agent` and receives events.

```
int main (void)
{
  // Create application resources for
  // logging and display.  Some events will
  // be logged to a database, while others
  // will be written to a console.
  Logger database_logger (DATABASE);
  Logger console_logger (CONSOLE);

  // Different graphical displays may need
  // to be updated depending on the event type.
  // For instance, the topology window showing
  // an iconic view of the system needs to be
  // updated when new connection events arrive.
  Window main_window (200, 200);
  Window topology_window (100, 20);

  // Create an ACT that will be returned when
  // connection events occur.
  EMIS_Event_Handler::State
    connection_act (&topology_window;
                    &database_logger);

  // Create an ACT that will be returned when
  // connection events occur.
  EMIS_Event_Handler::State
    image_transfer_act (&main_window,
                        &console_logger);

  // Object that will handle all incoming
  // EMIS events.
  EMIS_Event_Handler handler;

  // Binding to a remote Agent that
  // will call back the EMIS_Event_Handler
  // when EMIS events occur.
  Agent agent = ... // Bind to an Agent proxy.

  // Register with Agent to receive
  // notifications of EMIS connection events.
  agent.register (&handler,
                  Agent::NEW_CONNECTIONS,
                  reinterpret_cast <ACT>
                    &connection_act);

  // Register with Agent to receive
  // notifications of EMIS image transfer
  // events.
  agent.register (&handler,
                  Agent::IMAGE_TRANSFERS,
                  reinterpret_cast <ACT>
                    &image_transfer_act);

  run_event_loop ();
}
```

The application starts by creating its resources for logging and display. It then creates `State` objects that identify the completion of connection and image transfer events. The `State` objects contain references to `Window` and `Logger` objects. The address of the `State` objects are used as the ACTs. Next, the application registers the `Management Application` instance with the `Agent` for each type of event. Finally, the application enters its event loop, where all GUI and network processing is driven by callbacks.

When an event is generated by an EMIS component, the `Agent` sends the event to the `Management Application`, where it is delivered via the `recv_event` upcall. The `Management Application` then uses the ACT returned to access the state associated with the event. With the `State` object, it updates a graphical window and logs the event. Note that the `Management Application` uses ACTs to decide the appropriate action for each event. Image transfer events are displayed on a main window and logged to the console, whereas new connection events are displayed on a system topology window and logged to a database.

# 11 Variations

**Non-opaque ACTs:** In some implementations of the ACT pattern, Services do not treat the ACT as a purely opaque objects. For instance, Win32 `OVERLAPPED` structures are non-opaque ACTs because certain fields can be modified by the kernel. One solution to this problem is to pass subclasses of the `OVERLAPPED` structure that contain the additional state.

**Synchronous ACTs:** ACTs can also be used for operations that result in synchronous callbacks. In this case, the ACT is not really an *asynchronous* completion token, but a *synchronous* one. Using ACTs for synchronous callback operations provides a well-structured means of passing state related to the operation through the Service. It also maintains a decoupling of concurrency policies. Thus, the code which receives the ACT can be used for synchronous or asynchronous operations.

# 12 Known Uses

As described below, the Asynchronous Completion Token (ACT) pattern is widely used in many systems software and communication middleware, as well as in less technical domains, such as Fedex inventory tracking:

**Operating system asynchronous I/O mechanisms:** The Asynchronous Completion Token pattern is used by most operating systems that support asynchronous I/O. For instance, ACTs are used in conjunction with handles, Overlapped I/O, Win32 I/O completion ports on Windows NT [9]. When an application running on this platform performs multiple asynchronous operations, such as network and file I/O, the operation completion results are queued in an I/O completion port, where they can be retrieved by the `GetQueuedCompletionStatus` system call. Likewise, the POSIX Asynchronous I/O API [8] uses ACTs for its asynchronous I/O read and write operations. In the POSIX API, results can be dequeued through the `aio_wait` and `aio_suspend` interfaces, respectively.

**CORBA demultiplexing:** The TAO [10] CORBA Object Request Broker (ORB) uses the Asynchronous Completion Token pattern to demultiplex various types of requests and responses efficiently, scalably, and predictably. On the client-side, for example, TAO uses ACTs to associate responses from a server with the appropriate client thread that invoked the request. Each TAO client request carries a unique opaque sequence number, *i.e.*, the ACT, which is represented as a 32-bit integer. When an operation is invoked, the client-side TAO ORB assigns its sequence number to be an index into an internal connection table managed using the Leader/Followers pattern [11]. Each table entry keeps track of a client thread that's waiting for a response from its server. When the server replies, it returns the sequence number ACT sent by the client. TAO's client-side ORB uses the ACT to index into its connection table to determine which client thread to awaken with the reply.

Likewise, on the server-side, TAO uses the Asynchronous Completion Token pattern to provide a low-overhead demultiplexing throughout the various layers of features in an Object Adapter [12]. For instance, when a server creates an object reference, TAO Object Adapter stores special object ID and POA ID values in its object key, which is ultimately passed to clients as an ACT contained in an object reference. When the client passes back the object key with its request, TAO's Object Adapter extracts the special values from the ACT and uses them to index directly into tables it manages. This so-called *active demultiplexing* scheme ensures $O(1)$ lookup regardless of the number of objects in a POA or the number of nested POAs in an Object Adapter.

**EMIS network management:** The example described in this paper is derived from a distributed Electronic Medical Imaging System being developed at Washington University for Project Spectrum [1, 3]. A network management application monitors the performance and status of multiple components in an EMIS. Agents provide the asynchronous service of notifying the Management Application of EMIS events, such as *connection events* and *image transfer events*. Agents use the Asynchronous Completion Token pattern so that the management application can efficiently associate state with the arrival of events from agents that correspond to earlier asynchronous registration operations.

**FedEx inventory tracking:** One of the most intriguing examples of the Asynchronous Completion Token pattern is implemented by the inventory tracking mechanism used by Federal Express postal services. A FedEx Airbill contains a section labeled: "Your Internal Billing Reference Information (Optional: First 24 characters will appear on invoice)." The sender of a package uses this field as an ACT. This ACT is returned by FedEx (the Service) to you (the Client) with the invoice that notifies the sender that the transaction has completed. FedEx deliberately defines this field very loosely, *i.e.*, it is a maximum of 24 characters, which are otherwise "untyped." Therefore, senders can use the field in a variety of ways. For instance, a sender can populate this field with the index of a record for an internal database or with a name of a file containing a "to-do list" to be performed after the acknowledgement of the FedEx package delivery has been received.

# 13 Consequences

There are several advantages to using the Asynchronous Completion Token pattern:

**Relieves Clients from managing state explicitly:** Clients need not keep complicated records of pending events because the ACT returned on completion contains all the necessary information.

**Allows efficient state acquisition:** ACTs can be very time efficient because they do not require complex parsing of data returned with the notification. All relevant information about the event can be stored either in the ACT or in an object pointed to by the ACT. Alternatively, ACTs can be used as indices or pointers to operation state for highly efficient access, *e.g.*, y eliminating table lookups.

**Space efficient:** ACTs need not be large in order to provide applications with hooks for associating state with asynchronous operations. For example, in C and C++, ACTs that are four byte `void *`'s can serve reference large objects.

**Flexible:** User-defined ACTs are not forced to inherit from an interface in order to use the Service's ACTs. This allows applications to pass as ACTs objects for which changing the type is undesirable or even impossible. The typeless nature of ACTs can be used associate an object of any type to an asynchronous operation. When ACTs are implemented in C and C++ as `void *`'s, they can be type casted to pointers to any type needed.

**Separates Client policies from Service mechanisms:** ACTs are typically typeless objects. By using typeless ACTs, such as C/C++ `void *`'s, the ACT representation remains opaque to the Service and need not be type-intrusive to Clients.

**Does not dictate concurrency policies:** Long duration operations can be executed asynchronously because operation state can be efficiently recovered from the ACT. This allows Clients to be single-threaded or multi-threaded depending on application requirements. Alternatively, a Service that does not provide ACTs may force delay-sensitive Clients to perform operations synchronously within threads to handle operation completions properly.

There are several potential pitfalls to avoid when using the Asynchronous Completion Token pattern. These include:

**Memory leaks:** Memory leaks can result if a Client uses ACTs as pointers to dynamically allocated memory and a Service fails to return the ACT, if the Service crashes, for instance. Clients wary of this possibility should maintain separate ACT repositories or tables that can be used for explicit garbage collection in case services fail.

**Application remapping:** If ACTs are used as direct pointers to memory, errors can occur if part of the application is remapped in virtual memory. This is plausible in persistent applications that may be restarted after crashes, as well as for objects allocated out of a memory-mapped address space. To protect against these errors, indices to a repository can be used as ACTs. This extra level of indirection provides protection against remappings, because index values remain valid across remappings, whereas pointers to direct memory may not.

**Authentication:** When an ACT is returned to a Client upon completion of an asynchronous event, the Client may need to authenticate the ACT before using it. This is necessary if the server cannot be trusted to have treated the ACT opaquely and may have changed the value of the ACT.

# 14 See Also

An Asynchronous Completion Token is typically treated as a Memento [13] by the underlying framework. In the Memento pattern, Originators give Mementos to Caretakers who treat the Memento as "opaque" objects. In the ACT pattern, Clients give ACTs to Services that treat the ACTs as "opaque" objects. Thus, the ACT and Memento patterns are similar with respect to the participants. However, the patterns differ in motivation and applicability. The Memento pattern takes "snapshots" of object states, whereas the ACT pattern associates state with the completion of asynchronous operations.

# Acknowledgements

Thanks to Paul McKenney and Richard Toren for their insightful comments and contributions.

# References

[1] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," in *Proceedings of the* $2^{nd}$ *Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.

[2] P. Jain, S. Widoff, and D. C. Schmidt, "The Design and Performance of MedJava – Experience Developing Performance-Sensitive Distributed Applications with Java," *IEE/BCS Distributed Systems Engineering Journal*, 1998.

[3] G. Blaine, M. Boyd, and S. Crider, "Project Spectrum: Scalable Bandwidth for the BJC Health System," *HIMSS, Health Care Communications*, pp. 71–81, 1994.

[4] G. Parulkar, D. C. Schmidt, E. Kraemer, J. Turner, and A. Kantawala, "An Architecture for Monitoring, Visualization, and Control and Gigabit Networks," *IEEE Network*, vol. 11, September/October 1997.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[6] D. C. Schmidt and S. Vinoski, "Introduction to CORBA Messaging," *C++ Report*, vol. 10, November/December 1998.

[7] S. Berczuk, "A Pattern for Separating Assembly and Processing," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[8] "Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]," 1995.

[9] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[10] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[11] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 1999.

[12] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* $5^{th}$ *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.