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

This paper appeared in the Proceedings of the 3^{rd} annual Pattern Languages of Programming conference held at Allerton Park, Illinois, September 4-6, 1996.

Abstract

Contemporary applications must respond to many types of events, ranging from user interface notifications to network messages. Delay-sensitive applications, such as network management systems, often perform long-running operations asynchronously to avoid blocking the processing of other pending events. When these asynchronous operations complete, applications may need more information than simply the notification itself to properly handle the event. This paper describes the Asynchronous Completion Token pattern, which allows applications to efficiently associate state with the completion of asynchronous operations.

1 Intent

To efficiently associate state with the completion of asynchronous operations.

2 Also Known As

"Magic Cookie"

3 Motivation

3.1 Context

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.[1]

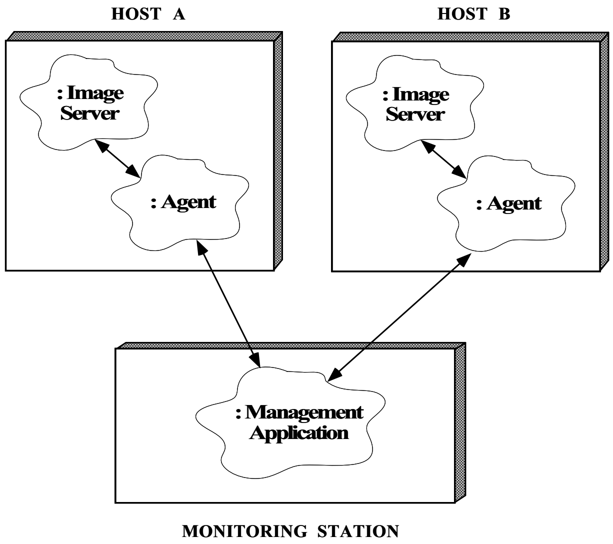

Figure 1: Participants in the EMIS Management System.

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 the Management Application at a central Monitoring Station. Administrators use the Management Application to view and control the overall status and performance of the EMIS.

Figure 2 shows a typical sequence of events between the Management Application, Agents, and EMIS components. The Management Application initially registers with an Agent for periodic updates. For example, an application may register to receive events every time an Image Server on a particular host accepts a new connection from a diagnostic workstation. When the Agent detects a new connection at the Image Server, it sends a "new connection" event to the Management Application. The Management Application can then display the new connection graphically on its console.

When an Agent sends an event to the Management Application several application-specific actions must be performed. For instance, the Management Application may need to update a graphical display or record the event in a

[1] There are many components in a distributed EMIS, including modalities, clinical and diagnostic workstations that perform imaging processing and display, hierarchical storage management (HSM) systems, and patient record databases. In this paper, we will consider only *Image Servers*, which store and retrieve medical images.

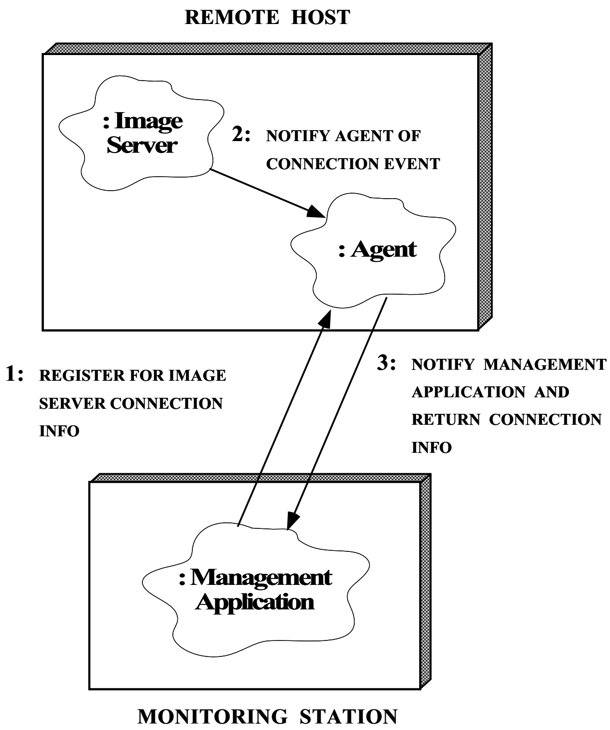

Figure 2: Event Registration and Callback.

local database. Since the Management Application can register for many types of events from any number of Agents, it must efficiently determine the appropriate action for each event. More specifically, Agents must provide a utility for the Management Application to associate additional state with an event notification. This associated state must be sufficient for the Management Application to determine the appropriate response (*e.g.,* displaying in a window or logging to a database) for the event.

In general, when providing an asynchronous service (*e.g.,* EMIS Agents propagating events to maintenance applications), the service must allow clients to associate state with asynchronous operations. Clients use the state to decide the appropriate actions to perform upon completion of the operations. The key design challenge examined in this paper how to efficiently associate application-specific state with the completion notifications.

3.2 Common Traps and Pitfalls

One way to execute multiple long duration operations simultaneously is to utilize threads. For instance, the Management Application could spawn a separate thread for each request to an Agent. Each request would execute synchronously, blocking until its Agent replied. In this approach, the state information required to handle the Agent replies could be stored implicitly in the context of each thread's run-time stack.

There are several drawbacks with a completely synchronous approach based on threading, however:

- *Increased complexity* – Threading may require complex concurrency control schemes;
- *Poor performance* – Threading may lead to poor performance due to context switching, synchronization, and data movement [2];
- *Lack of portability* – Threading may not be available on the OS platform.

Another way to associate state with the completion of an asynchronous operation is to depend on the information returned by the asynchronous service. For example, Agents must provide sufficient information in the callback so the Management Application can distinguish between all possible events. There are several limitations to this approach, however:

- *Excessive bandwidth* – Providing sufficient data to the client may require an excessive amount of data exchange. This may be particularly costly for distributed services.
- *Lack of context* – It may be too difficult for the service to know what data is needed by the clients. For instance, EMIS Agents cannot know the context in which the Management Application made the request.
- *Performance degradation* – The client may have to perform time consuming processing (*e.g.,* searching a large table) to uniquely identify the completion of an asynchronous event based on the data returned from the service. This extra processing can degrade the performance of the client.

3.3 Solution

Often, a more efficient and flexible way to associate state with completion events is to use the *Asynchronous Completion Token (ACT) pattern*. The ACT pattern resolves the following forces involved with associating state with asynchronous operations:

• **Time efficiency:** ACTs do not require complex parsing of data returned with the notification. Rather, the ACT can be an index or direct pointer to the necessary state.

• **Space efficiency:** ACTs do not need 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 as references to any size object.

• **Flexibility:** An ACT 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 cast to pointers to any type needed.

• **Non type-intrusive:** 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 event not possible.

To illustrate the solution more concretely, consider the EMIS example described in Section 3.1. When a Management Application registers with an Agent for periodic updates. it creates an ACT and passes it to the Agent. When an EMIS event occurs, the Agent calls the Management Application and passes back the ACT that was originally sent to it. Since the Agent does not change the value of the ACT, the Management Application can use the ACT to regain the state needed to process the event notification.

To make the example even more concrete, consider a typical scenario where an EMIS administrator uses the Management Application to log the connections made to a particular Image Server. As usual, the Management Application must register with the Image Server's Agent to be notified of connection events. However, when these connection events arrive at the Management Application, it must know to log the data in addition to performing its normal graphical updates.

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, as the ACT, a reference to a State object. The State object contains references to a user interface to be updated and a logging object. 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.

• **Does not force Concurrency Policies:** Figure 3 shows the order of events in such a scenario. Before the Management Application registers with the Agent, it creates the state needed to handle the event notifications. 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 is used to guide subsequent the event-specific actions.

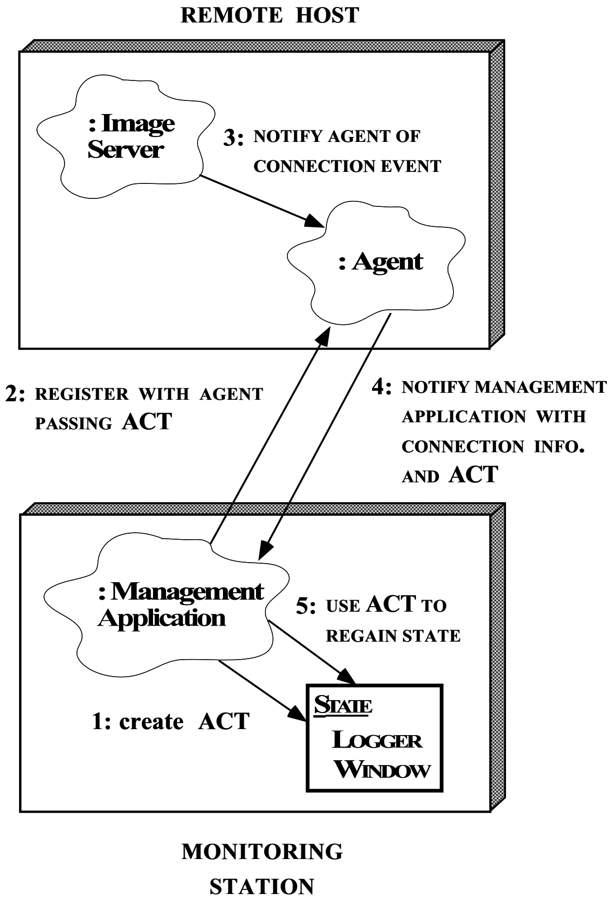

Figure 3: Event Registration and Callback with ACTs.

4 Applicability

Providers of asynchronous services should use the Asynchronous Completion Token Pattern under the following circumstances:

- The service performs client requests asynchronously. If the operations are synchronous, there may be no need to provide an explicit hook for regaining state since the state can be implicit in the activation record where the client blocks.
- The notification provided by the service upon the completion of an asynchronous operation is not sufficient for the clients to uniquely identify the operation.
- The service knows nothing about the application-specific nature of the clients. ACTs provide a clear separation between application *policy* and service *mechanism*. Since ACTs can be implemented without regard to type (*e.g.,* as `void *` pointers), services need not know anything about the type system used by clients.

5 Structure and Participants

Figure 4 illustrates the following participants in the Asynchronous Completion Token pattern:

- **Asynchronous Completion Token** (`Logger Window State`)
  - The Asynchronous Completion Token is given by Clients to Services to be returned on completion of asynchronous operations. Tokens can be indices into tables or direct pointers to memory holding the state necessary to handle the completion of the operation. However, to the Service the ACT is simply an opaque object that will not be read from or written to. In fact, ACTs are often represented as `void *`'s to avoid being type intrusive.
- **Service** (`Image Server Agent`)
  - The Service provides some type of asynchronous task to Clients. In the EMIS example, Agents

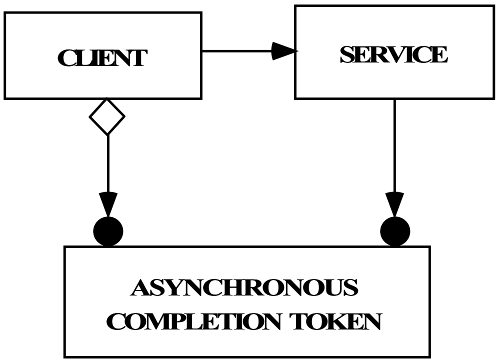

Figure 4: Structure of Participants in the Asynchronous Completion Token pattern.

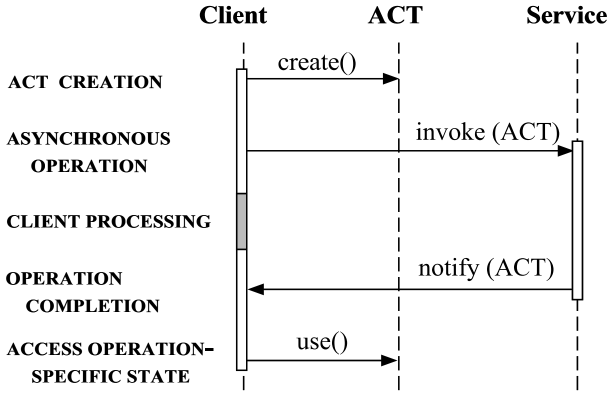

Figure 5: Participant Collaborations in the Asynchronous Completion Token Pattern.

provide a utility for asynchronously propagating EMIS events to Management Applications. Services may hold a collection of ACTs, the appropriate one of which will be returned to each Client when its asynchronous operation completes.

- **Client** (`Management Application`)
  - The Client performs requests for asynchronous operations on the Service. It requires application-specific state, along with the completion notification, to correctly handle asynchronous events. In the EMIS example, the Management Application is a Client to the Agent.

6 Collaborations

As shown in Figure 5, the interactions between the participants of the ACT pattern can be divided into the following phases:

- *ACT creation* – Before invoking an asynchronous operation on the Service, the Client creates the state associated with operation. This will be the ACT passed to the Service.
- *Asynchronous operation invocation* – When invoking the operation on the Service, the Client passes a reference to the state as an ACT.
- *Client processing* – The Client continues executing while the Service performs the request from the Client.
- *Asynchronous operation completion* – When the operation completes, the Service notifies the Client and returns the ACT. The Client uses the ACT to regain the needed state and continues with the application-specific action for that operation.

7 Consequences

7.1 Benefits

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 since the ACT returned on completion contains all the necessary information.
- *Allows efficient state acquisition* – ACTs can be used as indices or pointers to operation state for highly efficient access (*e.g.,* eliminates table lookups).
- *Simplifies client event handling algorithms* – Client code need not perform complicated processing on the returned data to handle the event. Thus, all relevant information about the event can be stored either in the ACT or in an object pointed to by the ACT.
- *Separates Client policies from Service mechanisms* – ACTs are typically typeless objects. By using typeless ACTs (*e.g.,* `void *`'s), the tokens remain opaque to the Service and are not type-intrusive to Clients.
- *Does not dictate concurrency policies* – Long duration operations can be executed asynchronously since 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.

7.2 Drawbacks

There are several potential pitfalls to avoid when using the Asynchronous Completion Token pattern, including:

- *Memory leaks* – If a Client uses ACTs as pointers to dynamically allocated memory and a Service fails to return the ACT (*e.g.,* the Service crashes) memory leaks can result. 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 memory-mapped space. To protect against these errors, indices to a repository can be used as ACTs. This extra level of indirection provides protection against remappings, since index values remain valid across remappings, whereas pointers to direct memory do not.

8 Implementation

From the viewpoint of the Service, there are four steps developers must follow to implement the ACT pattern: *(1) define the ACT representation*, *(2) accept the ACT* from a client, *(3) hold the ACT* while the operation is being performed, and *(4) return the ACT* to the client. The remainder of this section describes these steps.

8.1 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 languages like 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 OS platforms and compiler 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 uniform representation throughout the system.

When using a higher-level object-oriented language such as Java or Smalltalk, the developer may represent ACTs as references to abstract base classes. Since the Service typically can not make assumptions about the use of the ACTs, this class will be (mostly) empty (*e.g.*, a Java `Object`). Upon receiving ACTs, clients can use dynamic casting to narrow the ACT to a meaningful type.

8.2 Accept the ACT

This step is typically straightforward. ACTs can be passed as parameters to asynchronous (or synchronous) operation invocations. In section 9 we show an example of how ACTs can be parameters to Service methods.

8.3 Hold the ACT

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 of the Service. 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 asynchronous services to the Client.

However, 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, that, 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 but the last member in the chain are both Clients and Services since they 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.

In this case, a Service must manage the ACTs so they can be returned to the appropriate Clients when operations complete. For instance, an EMIS Agent can service multiple event registrations simultaneously. Internally, Agents utilize asynchronous services, namely timer events. As a result, the task of ACT *holding* becomes more difficult. Fortunately, the timer mechanism used by the Agent can also use the ACT pattern, *i.e.,* the Agent stores the Client ACTs in the timer ACTs. The Agent code is simplified since it leverages the ACT hooks provided by the timer mechanism. Sometimes, however, asynchronous Services must create tables to manage state explicitly.

8.4 Return the ACT

In the EMIS example explained above, when the Management Application initiated a request on an Agent, the ACT was returned once with response. However, depending on application requirements, ACTs from a single request may be returned multiple times. For instance, if a stream of responses are tagged with the same ACT, the receiving Client can use the ACT to associate additional state with the stream.

In both the single or stream responses, there are several options to *how* ACTs are returned to Clients when asynchronous operations complete. In other words, you must decide how Clients should be notified 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 [3]. 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 9 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 [4]. 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 since Clients need not explicitly wait for notifications, *e.g.,* they need not block in an event loop.

Consider the chain of Services example explained in Section 8.3. 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 since, 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.

9 Sample Code

The sample 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 async operations.
  struct State
  {
    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 = static_cast <State *> (act);

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

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

    // ...
  }
};
```

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

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

```
class Agent
{
  // 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 which 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,
                  (ACT) &connection_act);

  // Register with Agent to receive
  // notifications of EMIS image transfer
  // events.
  agent.register (&handler,
                  Agent::IMAGE_TRANSFERS,
                  (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.

10 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 since 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 (*i.e.,*, a "SCT"). 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.

11 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:

• **OS asynchronous I/O mechanisms:** ACTs can be found in Win32 handles, Win32 Overlapped I/O, and Win32 I/O completion ports [5], as well as in the POSIX Asynchronous I/O API [4]. When an application is performing multiple asynchronous operations (*e.g.,* network and file I/O), there is typically one location (such as a Win32 I/O completion port) where the operation completion results are queued. For UNIX and POSIX asynchronous I/O read and write operations, results can be dequeued through the `aio_wait` and `aio_suspend` interfaces, respectively.

• **RPC transaction identifiers:** Client-side stubs generated by Sun RPC [6] use ACTs to ensure that requests from a client match up with responses from the server. Every client request carries a unique opaque transaction ID (the ACT), which is represented as a 32-bit integer. This ID is initialized to a random value when the client handle is created and is changed every time a new RPC request is made. The server returns the transaction ID value sent by the client. Client routines test for a matching transaction ID before returning an RPC result to the application. This assures the client that the response corresponds to the request it made. SUN RPC is an example of the "non-opaque" variation of the ACT pattern. Although the server can test the ACT for equality (*e.g.,* to detect duplicates), it is not allowed to interpret the ACT any further.

• **EMIS network management:** The example described throughout this paper is derived from a distributed Electronic Medical Imaging System being developed for Project Spectrum [1, 7]. 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 (*e.g.* connection events and image transfer events). Agents use the ACT pattern so that the Management Application can efficiently associate state with the asynchronous arrival of EMIS events.

• **FedEx inventory tracking:** One of the most intriguing examples of Asynchronous Completion Tokens 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.

12 Related Patterns

An Asynchronous Completion Token is typically treated as a Momento [8] by the underlying framework. In the Momento pattern, Originators give Momentos[3] to Caretakers who treat the Momento as "opaque" objects. In the ACT pattern, Clients give ACTs to Services that treat the ACTs as "opaque" objects. Thus, the ACT and Momento patterns are very much similar with respect to the participants. However, the patterns differ in motivation and applicability. The Momento pattern takes "snapshots" of object states; the ACT pattern associates state with asynchronous operations. Though, at some level, the patterns seem to be exactly the same.

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] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[3] 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.

[4] "Information Technology – POSIX Realtime Extension (C Lanaguage)," Tech. Rep. P1003.1c/, 1995.

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

[6] Sun Microsystems, *Open Network Computing: Transport Independent RPC*, June 1995.

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

[3] Momentos are not to be confused with Mentos^{TM} breath fresheners.

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