Flexible IDL Compilation for Complex Communication Patterns

Eric Eide James L. Simister Tim Stack Jay Lepreau

University of Utah Department of Computer Science
50 South Central Campus Drive, Room 3190
Salt Lake City, Utah 84112–9205

Phone: +1 (801) 585–3271; FAX: +1 (801) 585–3743
`{eeide,simister,stack,lepreau}@cs.utah.edu`

`http://www.cs.utah.edu/flux/`

Abstract

Distributed applications are complex by nature, so it is essential that there be effective software development tools to aid in the construction of these programs. Commonplace "middleware" tools, however, often impose a tradeoff between programmer productivity and application performance. For instance, many CORBA IDL compilers generate code that is too slow for high-performance systems. More importantly, these compilers provide inadequate support for sophisticated patterns of communication. We believe that these problems can be overcome, thus making IDL compilers and similar middleware tools useful for a broader range of systems.

To this end we have implemented *Flick,* a flexible and optimizing IDL compiler, and are using it to produce specialized high-performance code for complex distributed applications. Flick can produce specially "decomposed" stubs that encapsulate different aspects of communication in separate functions, thus providing application programmers with fine-grain control over all messages. The design of our decomposed stubs was inspired by the requirements of a particular distributed application called Khazana, and in this paper we describe our experience to date in refitting Khazana with Flick-generated stubs. We believe that the special IDL compilation techniques developed for Khazana will be useful in other applications with similar communication requirements.

Keywords: Flick, IDL compiler, interface definition language, compilation, optimization, communication patterns, middleware, CORBA

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement number F30602–96–2–0269. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

To appear in a special issue of *Scientific Programming,* 1999.

1 Introduction

Distributed applications have inherently complex behaviors and requirements, and therefore, it is essential that there be software development tools — so-called *middleware* — to simplify the construction of distributed systems. Because there are many different kinds of middleware, an application designer must be careful to choose the most appropriate middleware for the development of each particular system. The ideal middleware for a particular programming task would be a tool (or set of tools) that simultaneously satisfies three criteria. First, the tool would minimize the human effort required to design, implement, and maintain the distributed application. Second, the tool would result in efficient and fast application code. Third, the tool would strongly support the application's overall design and programming model.

Unfortunately, for many high-performance applications, typical middleware systems do not meet all of these requirements simultaneously. For instance, typical interface definition language (IDL) compilers are often unsuitable for use in complex, high-performance distributed applications because they generate code that is unacceptably slow [18, 20, 23] and because they provide inadequate support for sophisticated programming models and fast communication infrastructures. We believe, however, that this need not be the case. We believe that the development of distributed applications can be improved through the use of high-level tools such as IDL compilers and that such middleware can be made to satisfy the demanding requirements of high-performance distributed applications, not only in terms of performance, but also in terms of support for complex programming models and communication patterns. To explore the design and use of such middleware, we have implemented *Flick*, the Flexible IDL Compiler Kit [9].

Flick provides a unique framework for experimenting with new IDL compilation techniques, and we are using Flick to develop new strategies for producing IDL-based stubs that meet the needs of sophisticated distributed applications. As an initial experiment in this area, we chose to generate specialized CORBA [24] stubs according to the requirements of a particular application called *Khazana* [5]. Khazana is an existing, complex, distributed application — a distributed memory system — that requires fine-grain control over the processing and handling of messages. We analyzed the communication patterns and implementation of Khazana and then created new Flick compiler components to generate specially "decomposed" stubs for use in this application. These stubs are "decomposed" because they separate the different aspects of communication into separate functions that encode, decode, transmit, and receive messages. (Normally, a single stub would encapsulate all of these functions.) We are currently modifying Khazana in order to replace the previous hand-coded stubs with the special stubs now being generated by Flick, and this paper details our experience to date in the development and application of our decomposed stubs.

We believe that Flick's ability to produce decomposed stubs will be useful not only for Khazana, but also for a wide range of distributed applications that require fine control over communication and asynchronous processing of messages by both clients and servers — applications that are poorly served by traditional IDL compilers. Flick's ability to produce optimized code for these applications demonstrates that high-level middleware can be successfully used in the development of complex applications.

2 IDL Compilers

An *interface definition language* (IDL) compiler is a tool that generates code to implement communication between separate software components. Given a high-level description of a software component, an IDL compiler produces special functions called *stubs*: functions that carry out communication between a *client* that invokes an operation and a *server* that implements the operation. Often, the client and server are located in separate processes, which may be running on separate machines. A stub encapsulates the details of communication and allows the client and server to interact through a procedural interface. Traditional IDL-based stubs encapsulate all aspects of communication — the encoding and decoding of data, the transmission and receipt of messages, and the handling of errors — and therefore, the stubs have the outward appearance of ordinary procedure calls. When this is the case, we say that the stubs implement *remote procedure call* (RPC) [3] semantics, or in an object-oriented language, that they implement *remote method invocation* (RMI) semantics.

IDL specifications are generally small and simple, and therefore, the use of IDL compilers can greatly reduce the human effort required to produce the communication code for a distributed application. This means that IDL compilers generally meet the first criterion for successful middleware. Unfortunately, for many high-performance applications, they often fail to meet the second and third.

Many common IDL compilers such as `rpcgen` [30] fail to meet the second criterion because they generate inefficient code — code containing excessive function calls and redundant runtime tests [23]. Gokhale and Schmidt [18] and others have quantified this and similar types of overhead within several CORBA implementations. If a distributed application makes frequent RPC calls and communicates across a sufficiently fast network, then the overhead within IDL-generated code can be a serious barrier to performance [20]. For this reason, the designers of high-performance distributed and parallel applications have been unwilling to make use of high-level tools such as IDL compilers that ease the creation of communication code.

The architects of complex distributed applications are further dissuaded from using IDL-based tools because typically, these tools fail to meet the third criterion for successful middleware: strong support for the desired application programming models. Many IDL compilers support only a single communication model — synchronous RPC — and this model is simply inappropriate for many distributed applications. Some IDL-based middleware systems support additional communication models through the use of runtime "services": libraries of objects that act as communication proxies and which implement new application-level communication models atop compiler-generated synchronous RPC stubs. The CORBA Event Service [25] is an example of this approach, as is CORBA's support for "deferred synchronous" communication through the Dynamic Invocation Interface [24].[1] These services, however, do not necessarily provide sufficiently strong support to high-performance applications. The communication proxies introduce new overheads [19] and applications must be specially written to communicate through these proxy interfaces. To avoid these problems, a middleware system like CORBA must support a variety of communication models not as run-time services, but as compile-time code generation options. The importance of compiler support has recently been highlighted by the adoption of the CORBA Messaging Specification [26], which defines a new standard for creating asynchronous stubs from CORBA IDL instead of the usual synchronous stubs.[2] This new specification is a significant step in broadening the usefulness of CORBA middleware, but it still falls short for many high-performance applications. Most notably, the Messaging Specification provides asynchronous stubs only for clients; servers must still process requests as if they were synchronous procedure calls.

3 Flick

Common IDL compilers are designed specifically to support a single IDL, a small, hardwired set of target language bindings (e.g., the standard mapping from CORBA IDL to C), and generally, one or at most a few message formats and transport facilities. In other words, these compilers are "rigid" and not easily adapted or enhanced. Flick, on the other hand, was designed from the start to be an extremely flexible and extensible IDL compilation framework.

Flick, the Flexible IDL Compiler Kit, is a set of compiler components that may be specialized to generate code for particular IDLs, stub styles, and transport systems. The overall structure of Flick is shown in Figure 1. Flick incorporates design principles from traditional language compilers and, like most modern compilers, divides compilation into three separate stages: front end parsing, intermediate code generation, and back end code optimization. Each stage has several implementations: for example, there are three separate front ends that parse the CORBA, ONC RPC (a.k.a. Sun RPC) [29], and MIG [27] IDLs. The different compiler stages communicate through intermediary files. The clean separation between stages makes it easier for compiler authors to implement new Flick components, for instance, in order to generate new kinds of stubs or to support new transport systems. Flick further eases the implementation of new compiler components by providing a large base library for each stage of compilation. For instance, because Flick's back end library implements many common stub optimization strategies, all of Flick's specific back ends automatically inherit these optimizations.

A *front end* reads an IDL specification and produces an Abstract Object Interface (AOI) file containing the parsed interface description (i.e., a parse tree). As just described, Flick has three separate front ends for parsing the CORBA, ONC RPC, and MIG IDLs. Each of these front ends is a component, built upon a large library of code that implements common functions — for example, functions to manipulate AOI parse trees and other intermediate data structures. Most IDL compilers, including those based on the SunSoft CORBA IDL compiler front end [31], are designed to produce code directly from an IDL parse tree. These compilers therefore provide little infrastructure for compiler writers to reuse when designing new stub styles (i.e., language bindings) or when implementing new code optimizations. Flick, on the other hand, provides reusable infrastructure for both of these compilation steps.

An intermediate code generator — called a *presentation generator* — reads an AOI file and then determines the application programmer's interface to the stubs: i.e., everything that an application programmer must know in order to use the stubs that will ultimately be produced by Flick. The programmer's view of the stubs — called a *presentation* — includes not only the names of the generated stubs and the types of their arguments, but also such things as the purpose of each stub, the stubs' calling conventions, the stubs' memory allocation conventions, and so on. Obviously, there is more than one way to map an IDL specification onto functions and other constructs in a programming language such as C; for example, an IDL compiler might produce synchronous or asynchronous stubs (or both). Because there are multiple ways in which one might present a single IDL interface, Flick provides multiple presentation generators for creating different kinds of stubs. As shown in Figure 1, Flick includes presentation generators for CORBA-, `rpcgen`-, Fluke- [14], and MIG-style C language stubs. Moreover, Flick provides the infrastructure for producing altogether new kinds of stubs, either through the creation of new presentation generators or through extensions to Flick's base presentation generator library. The output of a presentation generator is another intermediate file, called a Presentation in C (PRES_C) file.

A *back end* reads the stub descriptions contained in a PRES_C file and produces the optimized (C language) source code for the stubs. Because a PRES_C file describes *only* the programmer-visible appearance and behavior of the stubs, different back ends may be used to implement the PRES_C-described stubs atop different underlying transport systems and message formats. A single PRES_C description can be implemented using any suitable transport, and Flick currently provides back ends that implement client/server communication using CORBA IIOP, ONC/TCP, Mach [1] messages, Trapeze [2] messages, and Fluke kernel IPC. Each back end is specialized for a particular transport facility and message format, but they all incorporate the numerous optimization techniques provided by Flick's back end library. For instance, the library allows each back end to analyze the overall storage requirements of every message in order to produce efficient message buffer management code that is free from superfluous run-time checks and other typical run-time overheads. The library also analyzes the representations of message data in order to produce efficient marshaling and unmarshaling code. For example, a Flick-generated stub may use `memcpy` to copy an object if Flick is able to determine that no byte-swapping or other presentation layer data transformations are required. Previously reported experiments [9] demonstrate that Flick-generated stubs can marshal data between 2 and 17 times as fast as equivalent stubs generated by other IDL compilers. The reduction in presentation layer overhead results in significant end-to-end throughput improvements for communicating applications.

Because Flick is both flexible and optimizing, it provides an excellent basis for experimenting with new IDL compilation techniques: new IDLs, new presentations (stub styles), and new transport facilities. We are now leveraging Flick's infrastructure to develop new IDL compilation techniques to support applications with complex communication requirements, and as an initial experiment in this domain, we have extended Flick to create specialized stubs according to the needs of a particular distributed application called Khazana.

4 Khazana

Khazana is a distributed "global memory service" in development by Carter et al. at the University of Utah [5]. The Khazana system provides a single, globally shared, persistent storage space for distributed applications. The primary goals of Khazana are to provide robust, scalable, and efficient storage while also providing flexibility through "hooks" that allow higher-level services and applications to tailor Khazana's behavior to their needs. For instance, an application can specify the degree of replication required for its data and can specify how its distributed data should be kept consistent. The Khazana system provides only the base operations for managing distributed storage, leaving higher-level semantics to other middleware layers or to the applications themselves.

Khazana is implemented as a collection of peer processes that collectively maintain a single, flat, global memory space. The processes exchange messages in order to implement the Khazana protocol, which provides operations to:

- `reserve` and `unreserve` a contiguous region of the global address space;
- `allocate` and `free` physical storage for a previously reserved region of the global address space;
- `lock` and `unlock` parts of a region in various modes (e.g., read-write or read-only);
- `read` and `write` parts of a region; and
- `get` and `set` the attributes of a region, which specify such properties as the required consistency model and level of replication.

Although there are conceptual request and reply messages in a Khazana system, there are no specially designated "server" or "client" nodes. Rather, the Khazana protocols require every process to act as both client and server — i.e., every node must be able to transmit and receive both requests and replies as necessary. Further, the Khazana protocol requires that each process be able to participate in several operations at the same time. A node that transmits a request must be able to handle incoming messages before the reply to the original request is received. In processing a request, a Khazana node may discover that it must forward the request to another node (e.g., the home node of some requested memory page). Alternately, a node may discover that it can only partially service a request because some needed resource (e.g., a memory page) is not currently available. In that case, the request must be "pushed back" onto the queue of incoming requests along with the partial results that have so far been computed. Not all Khazana messages can be described in terms of isolated request/reply pairs. For instance, a single `allocate` request may result in multiple reply messages, each satisfying a portion of the original request. Similarly, a client `write` request will cause the server to issue requests back to the client for segments of the data. Only after all the data has been collected does the server reply to the original `write` request.

Khazana was originally implemented using hand-coded functions to process the system's protocols via TCP. Every node maintained a work queue; every element in the queue contained both a message and the contextual data representing the state of the message's processing (i.e., the protocol state of the Khazana operation that involves the message contained in the work queue node). Because a single structure was used to contain both stub-level message data and high-level application data, the protocol processing functions were written to handle both levels of detail. This lack of abstraction between layers complicated the overall Khazana code, and ultimately, the use of low-level abstractions led to code that was burdensome to create, debug, and maintain.

The Khazana implementors expect that as development continues, the problems associated with hand-coded messaging stubs will only become more severe. More message types will be added to Khazana, and Khazana will be ported to new computer and network architectures. Each of these developments will increase the burden of maintaining all the messaging code by hand. Therefore, the Khazana system designers are interested in using Flick to generate Khazana's communication stubs, both to ease future development and to simplify the existing Khazana implementation.

5 Decomposed IDL-Based Communication

Using Flick, we have designed and implemented a new compilation style for CORBA IDL specifications that breaks apart traditional RPC stubs in order to support applications like Khazana that require:

- the asynchronous handling of messages by both clients and servers;
- efficient message encoding and decoding, including the ability to cache frequently used messages;
- message forwarding;
- the handling of multiple replies to a single request; and
- the ability to suspend and resume the processing of a message.

Whereas a traditional RPC stub encapsulates many different aspects of communication — encoding and sending the request, then receiving and decoding the reply — Flick generates code for Khazana in a style that encapsulates each aspect of communication within its own stub. This new "presentation style" for IDL-based interfaces is illustrated in Figure 2 and consists of:

- *marshaling stubs*, to encode request and reply messages;
- *unmarshaling stubs*, to decode request and reply messages;
- *send stubs*, to transmit marshaled messages to other nodes;
- *server work functions*, to handle received requests;
- *client work functions*, to handle received replies; and
- *continuation stubs*, to postpone the processing of messages.

The stub and function prototypes generated by Flick for this new "decomposed" presentation style are summarized in Table 1.

5.1 Marshaling and Unmarshaling Stubs

Each operation in the IDL specification results in three marshaling stubs: one to marshal operation requests, a second to marshal ordinary operation replies, and a third to marshal exceptional (error-signaling) replies. A marshaling stub takes as parameters the data to be encoded, followed by a standard CORBA environment parameter used to communicate exceptions back to the caller (i.e., errors that occur as the message is being encoded). The marshaled message is returned by the stub so that the application can send the message at a later time; that is, the encoding and transmission events are decoupled. Note that the type of the marshaled message is specific to a single interface (object type); this helps prevent programming errors in which a request or reply is sent to an object of an inappropriate type. Also note that the marshaled message is opaque. This allows each of Flick's back ends to choose a message format, data encoding, and implementation that is best suited to the application's requirements and the underlying transport facility (e.g., CORBA IIOP messages, or some more specialized system).

Each IDL-defined operation also results in two unmarshaling stubs: one to unmarshal requests and one to unmarshal replies (ordinary or exceptional). An unmarshaling stub takes a message, parameters into which the message data will be decoded, and a CORBA environment parameter that allows the unmarshaling stub to report errors to the application.

Marshaling and unmarshaling stubs provide applications with greater control over the handling of messages and enable certain application-specific optimizations that are not possible within a traditional RPC model. For example, a message can be encoded once and then sent to multiple targets; also, common replies can be premarshaled and cached, thus reducing response times. An especially useful optimization for Khazana is that a message can be redirected to another node without the overhead of decoding and re-encoding the message (assuming that the message data is not needed in order to make the forwarding decision).

5.2 Send Stubs

Once a request or reply message has been marshaled, an application transmits the message by invoking the appropriate send stub. Two send stubs are defined for each CORBA interface (object type): one for requests and another for replies. Unlike a traditional RPC stub, a send stub returns immediately after the message has been sent; it does not wait for a reply.

Also unlike a traditional RPC stub, the send stubs produced by Flick take two special parameters as shown in Table 1: an *invocation identifier* and a *client reference*. These parameters are unique to Flick's "decomposed" stub presentation style and are not standard CORBA stub parameters. An invocation identifier (`Invocation_id`) corresponds to a single message transmission event and is used to connect a reply with its associated request. The application is responsible for providing invocation identifiers to the send stubs; this allows for application-specific optimizations in the generation and management of the identifiers. A client reference (`Client`) is a CORBA pseudo-object [24], managed by the communications runtime layer, that provides additional contextual information for handling requests and replies. A client reference serves two primary functions. First, it locates the entity that will receive the reply for a given request. Keeping explicit client references makes it possible, for instance, for one node to forward a request message to another node and direct the eventual reply back to the original requester. Second, the client reference allows the application to save state from the time a request is sent until the corresponding reply is received. (This extra application data is not transmitted as part of a request or reply; it is simply stored by our CORBA runtime as a convenience to the application.) A process can allocate as many client references as it needs and associate them with request transmission events as it sees fit. Then, whenever a reply message is received, the runtime locates the client reference that was provided with the original request, and gives that reference to the function that will process the newly received reply.

The `Invocation_id` and `Client` parameters in Flick's decomposed presentation style are similar in purpose to the `ReplyHandler` objects defined by the recently adopted CORBA Messaging Specification [26]. Both a `Client` and a `ReplyHandler` can be used to hold application state. The primary difference between a `Client` and a `ReplyHandler` is in the set of operations that they support. Within the CORBA Messaging presentation, a reply to an asynchronous request is treated as a request on a `ReplyHandler` object; among other things, this means that the reply data will be unmarshaled before the `ReplyHandler` is invoked. Flick's decomposed presentation style, however, separates the receipt and decoding phases of reply processing, thus enabling additional flexibility. For example, a work function can forward a reply message from one `Client` to another without any intervening unmarshaling.

5.3 Work Functions

The server and client work functions are the functions that ultimately receive and service request and reply messages, respectively. Traditional RPC presentations contain server work functions only; traditional clients process requests and replies synchronously. In contrast, Flick's decomposed stub presentation allows replies to be handled asynchronously, and therefore, clients must contain explicit functions to receive and handle incoming replies. A server work function receives the marshaled representation of the request that it is to service. This makes it straightforward and efficient for a Khazana node to forward the request to another node, or to delay local processing. If the request is to be handled immediately, the server work function will invoke the unmarshaling stub to extract the request data. Similarly, a client work function receives a marshaled reply message so that it may choose to handle the reply immediately, forward the reply (to another client), or delay processing by invoking a continuation stub.

5.4 Continuation Stubs and Functions

While handling a request or reply, a client or server work function may need to postpone processing — for instance, a server handling a request may find that it needs data from another server before it can continue. To allow applications to handle these situations gracefully, Flick produces two "continuation" stubs for each operation: one for the request and another for the reply. These are essentially normal send stubs, except that they direct the message back into the node's own message queue. Each continuation stub accepts a message, a pointer to the function that will be invoked to service the continued message (i.e., a special work function), and an opaque pointer that allows the application to pass arbitrary state information along to the function that will resume processing. A separate runtime library function is provided to allow applications to "wake up" a continued message, thus allowing the message to be dispatched to the continuation work function.

6 Evaluation

In this section we describe our experience to date in refitting Khazana with Flick-generated, decomposed communication stubs. Although our extensions to Flick are complete, our modifications to Khazana are currently in progress.

6.1 Implementation Issues

Khazana's original communication substrate was entirely hand-coded and very specific to the Khazana application. The original routines used a single data structure to store both low-level and high-level information such as encoded message data (both input and output), message queue links, operation context data (e.g., locks), and application state (e.g., a `jmp_buf` so that Khazana could suspend message processing by executing a `longjmp` back to its message dispatch loop). This design was chosen for its initial ease of implementation, and these message structures were routinely passed from function to function during message processing. Both high-level and low-level processing occurred at many points in the code.

The use of decomposed stubs, however, mandated a much stricter application design. In keeping with the CORBA spirit, Flick-generated stubs require that nodes (represented as CORBA objects), client references, and messages all be opaque data structures, disallowing access from the application into low-level data such as socket file descriptors and node IP addresses. To reverse-engineer the CORBA IDL description of the Khazana protocols, we had to dissect the Khazana code and separate the different levels of information contained in Khazana's original message data structures. Data that was previously passed implicitly as part of a message (e.g., operation state and context) was now required to be handled explicitly in the IDL. Other state information, not properly part of the protocol messages, was moved into separate structures and managed through `Client` references (as described in Section 5.2) or continuation function data (Section 5.4). The introduction of explicit layers into Khazana's code has helped to clarify the application code and to document the Khazana communication protocols, which are now described in CORBA IDL. As a result of our application restructuring, we found it necessary to replace Khazana's original IPC code with our own CORBA ORB runtime. Although we had planned to leverage as much of Khazana's original IPC code as possible, this approach ultimately proved to be impractical, and we are now implementing a completely new CORBA runtime with support for both decomposed and traditional CORBA stubs. This new runtime will be used not only by Khazana but also by other applications that are written or modified to use Flick-generated stubs.

6.2 Communication Patterns

Although we have greatly modified Khazana in order to introduce structure, we have not had to modify Khazana's overall programming model. Our experience in applying Flick's decomposed stubs to Khazana has been positive, and we have been able to reimplement most of Khazana's communication patterns using our specially generated stubs. One particularly interesting pattern, previously described in Section 4, occurs as part of the `allocate` operation.

The `allocate` operation allows one node to ask another to reserve physical storage. If the receiver of the `allocate` message cannot satisfy the entire request, it will satisfy what it can, send a reply back to the requesting node describing the partial allocation, and then issue a new request message to another node on behalf of the original requester, asking this new node to fulfill the remaining portion of the allocation. This new node handles the request in a similar fashion, possibly sending new requests to yet more Khazana nodes. Thus, from the perspective of the original requester, there are many (partial) replies, coming from many different nodes, to its original allocation request. This communication model is directly supported by Flick's decomposed stubs. The requesting client indicates to the CORBA runtime that it expects to receive multiple replies to a particular request message. (The client must indicate this explicitly; otherwise, the ORB will release certain message-related data structures after the first reply is received and processed.) The decomposed stubs allow the server to both reply to the client's request and to manufacture a new request message on behalf of the client, using the requester's `Client` reference and message `Invocation_id`. The client's reply work function (Section 5.3) is invoked once for each partial reply and is responsible for aggregating the results. Finally, when the client determines that all replies have been received, it invokes an ORB function to indicate that no more replies to the original request are expected. In summary, although the notion of multiple replies to a single request is not expressed in CORBA IDL, Flick's decomposed stubs make it possible for an application to use this communication pattern (given the necessary support from the ORB runtime as described).

One Khazana idiom not directly supported by the decomposed stubs was the piecemeal provision of message data to the underlying IPC facility. The original hand-coded IPC system allowed Khazana to send a message by providing a callback function: whenever the IPC layer was ready, it would invoke the callback to acquire the next block of message data. This feature allowed Khazana to send large messages without requiring that all of the data be in memory at once. On the receiving node, special code was used to process such large messages. Because the IPC system exposed the underlying socket file descriptor, the Khazana application could read and process message data in a similarly piecemeal fashion. Our decomposed stubs, however, do not directly support this kind of incremental message processing. Instead, all message data must be provided when a message is marshaled by a send stub (Section 5.2), and all message data must be received before the ORB will dispatch a message to the Khazana application. Fortunately, it is straightforward to emulate the old communication idiom with decomposed stubs, simply by breaking the previous single, large message into multiple, smaller messages. On the sending node, message transmission can be paced through the use of special ORB runtime functions that notify the Khazana application that particular message instances have been sent.

Finally, the use of decomposed stubs had important implications for messages sent to local objects — i.e., objects that are in the same address space as the client. Through traditional synchronous RPC stubs, when a server and client are co-located, communication can transparently take place without expensive marshaling and unmarshaling of message data. In Khazana, this is particularly important for the `read` and `write` operations, which normally involve multiple messages and may pass large amounts of data. Much of this communication overhead can be avoided during local operations, and Khazana's original implementation contained special checks for optimizing communication with local objects. We found that we had to introduce similar special-purpose code when refitting Khazana to use our decomposed CORBA stubs. Because our decomposed stubs separate message encoding from message transmission, a Flick-generated marshal stub (Section 5.1) cannot know that a message will be dispatched to a local object. Therefore, to optimize local communication, our new Khazana implementation contains special checks and avoids using decomposed stubs for local objects. In future work we expect to modify the implementation of Flick's decomposed stubs to improve support for local object invocations, e.g., by delaying actual message marshaling until the first time a message is sent to a remote object.

6.3 Summary

Flick's decomposed stubs provide an appropriate communication abstraction for Khazana. Due to Khazana's initial implementation, we were forced to make significant modifications to the application in order to introduce our decomposed stubs and CORBA ORB runtime. However, we were able to make these modifications without changing Khazana's overall programming model. Our modifications to Khazana have made the application portable to new architectures and transport systems. Finally, our experience with decomposed stubs has highlighted the importance of ORB support for certain aspects of decomposed communication including (1) multiple replies to a single request, (2) notification of message transmission events, for application-level message pacing, and (3) optimizations for local objects.

We expect that our new, decomposed presentation style for CORBA interfaces will be useful not only to Khazana but also to other, similar distributed applications. Since this new style is more like traditional message-passing, we believe that it will be useful as a migration path for applications that wish to move away from hand-written communication code and toward the use of IDL-based middleware tools. This will reduce the application writers' burden and eliminate what is currently an error-prone process, and we expect, without compromising application performance.

7 Related Work

The CORBA Messaging Specification [26] is the Object Management Group's recently adopted standard for asynchronous messaging. That document describes the new standard method for mapping CORBA IDL onto asynchronous stubs, and the standard differs from Flick's decomposed stub presentation in several respects. For application writers, the most important difference is that Flick enables asynchronous message processing for both clients and servers, while the Messaging Specification defines asynchronous processing for clients only. Additionally, Flick's decomposed stubs allow applications to separate message encoding and decoding from message transmission and receipt. This allows applications to forward or cache messages for more efficient operation. Beyond the details of the generated stubs, however, the CORBA Messaging Specification is more broad than the work presented here. In addition to defining a standard for asynchronous stubs, the Messaging Specification defines a new Quality of Service (QoS) standard for CORBA-based applications, supported by a new message routing infrastructure. These new CORBA facilities are runtime components, and therefore outside the domain controlled by an IDL compiler like Flick.

Flick and the CORBA Messaging Specification represent two different approaches to flexible stub generation. The Messaging Specification increases flexibility by expanding the set of standard ways in which IDL may be translated into stubs. Flick, however, increases flexibility by opening the IDL compiler itself to new components, which may be developed rapidly and independently to target specific application domains. In previous work [12, 13] we demonstrated the benefits that come from the ability to create application-specific stubs. We showed that application-specific stubs, customized according to a set of programmer-supplied interface annotations, could provide up to an order of magnitude speedup in RPC performance.

The CORBA Event Service [25] is another Object Management Group standard for decoupling requests and replies between CORBA objects. This specification defines an *event channel* as an object that mediates communication between sets of suppliers and consumers. Because an event channel is a heavyweight object, it can provide many services — but these extra services may come at a price. To make use of any channel services, including asynchronous messaging, clients and servers must be specially written to communicate through event channels. This is in contrast to Flick's decomposed stubs which allow a client or server (or both) to use asynchronous messaging without cooperation or knowledge from the other side.[3] Also, because event channels interpose on communication, they may introduce overheads that are not present in Flick's optimized stubs.

The importance of optimizing middleware will only increase as computers and networks become increasingly fast. Modern operating systems are now supporting efficient, lightweight communication mechanisms such as shared memory-based intra-node communication channels [1], highly optimized kernel IPC paths [10, 21], and new inter-node communication models such as active messages [33] and sender-based protocols [4, 32]. As Clark and Tennenhouse predicted in 1990 [7], these improvements in low-level communication systems have moved the performance bottlenecks for distributed applications out of the network and operating system layers and into the applications themselves.

Recent work by Schmidt et al. [18, 28] has quantified the impact of presentation layer overhead for `rpcgen` and two commercial CORBA implementations. On average, due to inefficiencies at the presentation and transport layers, compiler-generated stubs achieved only 16–80% of the throughput of hand-coded stubs. To address these and similar performance issues, several attempts have been made to improve the code generated by IDL compilers. These are discussed in our earlier paper on Flick [9].

In summary, these other IDL compilers are either not very flexible (e.g., Mach's MIG compiler [27]) or not able to produce very fast code.

Asynchronous communication and message forwarding are not new ideas. Anderson et al. [2] describe these same mechanisms for a Global Memory Service [11] built on top of Trapeze. However, their work focuses on the transport layer rather than the presentation and application layers. Further, they provide no support for automatically generating stubs to exploit asynchronous communication and message forwarding. Our work focuses on presenting these mechanisms to the application and automatically generating the appropriate stubs. We believe that our work is complementary to that of Anderson et al.; Flick can leverage the benefits of an efficient transport system to produce optimized communication stubs.

For parallel applications, there are a large number of specialized programming languages such as CC++ [6], Fortran M [15], and Split-C [8]. In most of these cases the language handles marshaling and unmarshaling of parameters. However, it is our belief that the techniques used by Flick, and possibly even its code, could be incorporated into the compilers for these languages to substantially reduce presentation layer costs, e.g., by minimizing data copying. There are also a large number of parallel runtime systems providing various levels of abstraction and functionality, such as MPI [22], PVM [17], and Nexus [16]. Typically, these systems require the programmer to write the marshaling code by hand, although they do abstract away byte-swapping in order to accommodate heterogeneous machines. We believe these provide an attractive target for optimizations provided by Flick.

8 Conclusion

High-level tools such as IDL compilers can greatly reduce the effort and time required to implement distributed and parallel systems. Unfortunately, the limitations of traditional IDL compilers often prevent the use of such tools in applications that require maximum performance. Traditional IDL compilers may produce stubs that have excessive presentation layer overheads; furthermore, traditional RPC stubs are not a good match for the needs of many modern systems.

We have outlined Flick, a novel, modular, and flexible IDL compiler that generates optimized code for a variety of stub styles and transport mechanisms. We describe how Flick has been extended to meet the needs of a particular distributed application, Khazana, and now provides a new style of CORBA-based stubs appropriate for use in other similar systems. We believe that the creation and maintenance of complex distributed applications can be greatly improved through the use of middleware tools like Flick that minimize programmer effort, produce optimized code, and provide support for a flexible set of application programming models.

Availability

Complete Flick source code and documentation are available from `http://www.cs.utah.edu/flux/flick/`.

Acknowledgments

We wish to thank the Khazana implementors, especially John Carter, Sai Susarla, and Anand Ranganathan, for their help in this work. Without their support, this work would not have been possible. We also wish to thank the anonymous reviewers who made many helpful comments on previous drafts of this paper. Finally, we owe special thanks to Patrick Tullmann for providing extensive proofreading and suggestions for improvement.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, June 1986.

[2] D. C. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 143–154, New Orleans, LA, July 1998.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), Feb. 1984.

[4] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 245–259, Seattle, WA, Oct. 1996. USENIX Association.

[5] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An infrastructure for building distributed services. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 562–571, May 1998.

[6] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. Technical Report CS-TR-92-01, California Institute of Technology, Mar. 1993.

[7] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, 1990.

[8] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Portland, OR, Nov. 1993.

[9] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–56, Las Vegas, NV, June 1997.

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[11] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 201–212, Copper Mountain, CO, Dec. 1995.

[12] B. Ford, M. Hibler, and J. Lepreau. Using annotated interface definitions to optimize RPC. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, page 232, 1995. Poster.

[13] B. Ford, M. Hibler, and J. Lepreau. Using annotated interface definitions to optimize RPC. Technical Report UUCS-95-014, University of Utah Department of Computer Science, Mar. 1995.

[14] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, Oct. 1996. USENIX Assoc.

[15] I. T. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 25(1), Feb. 1995.

[16] I. T. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proceedings of First International Workshop on Parallel Processing*, pages 457–462, 1994.

[17] G. Geist and V. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogenous network of workstations. In *Sixth Annual Distributed-Memory Computer Conference*, pages 258–261, 1991.

[18] A. Gokhale and D. C. Schmidt. Measuring the performance of communication middleware on high-speed networks. *Computer Communication Review*, 26(4), Oct. 1996.

[19] A. Gokhale and D. C. Schmidt. The performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over high-speed ATM networks. In *Proceedings of GLOBECOM '96*, pages 50–56, London, England, Nov. 1996.

[20] A. Gokhale and D. C. Schmidt. Optimizing the performance of the CORBA Internet Inter-ORB Protocol over ATM. Technical Report WUCS–97–09, Washington University Department of Computer Science, St. Louis, MO, 1997.

[21] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[22] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. `http://www.mpi-forum.org/`.

[23] G. Muller, R. Marlet, E.-N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, May 1998.

[24] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1995.

[25] Object Management Group. Event service specification. In *CORBAservices Specification*, chapter 4. Object Management Group, Dec. 1997.

[26] Object Management Group. *CORBA Messaging: Joint Revised Submission with Errata*, May 1998. OMG TC Document orbos/98–05–06. `ftp://ftp.omg.org/pub/docs/orbos/98-05-06.ps`.

[27] Open Software Foundation and Carnegie Mellon University, Cambridge, MA. *Mach 3 Server Writer's Guide*, Jan. 1992.

[28] D. C. Schmidt, T. Harrison, and E. Al-Shaer. Object-oriented components for high-speed network programming. In *Proceedings of the First Conference on Object-Oriented Technologies and Systems*, Monterey, CA, June 1995. USENIX Assoc.

[29] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. Technical Report RFC 1831, Sun Microsystems, Inc., Aug. 1995.

[30] Sun Microsystems, Inc. *ONC+ Developer's Guide*, Nov. 1995.

[31] SunSoft, Inc. SunSoft OMG Interface Definition Language Compiler Front End, release 1.3, Mar. 1994. `ftp://ftp.omg.org/pub/contrib/OMG_IDL_CFE_1.3/`.

[32] M. R. Swanson and L. B. Stoller. Direct Deposit: A basic user-level protocol for carpet clusters. Technical Report UUCS-95-003, University of Utah Department of Computer Science, Mar. 1995.

[33] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

Notes

[1]Using the "deferred synchronous" communication model, a CORBA client sends a request message and later must poll for the corresponding reply. The server still handles the request synchronously. In CORBA, deferred synchronous communication is available only through the Dynamic Invocation Interface, which generally imposes significant communication overhead [19] and requires application programmers to write a significant amount of messaging code, thus significantly reducing the principal benefits that come from using an IDL-based middleware system.

[2]The CORBA Messaging Specification is not yet widely implemented.

[3]Some features of decomposed stubs, such as the ability to send multiple replies to a single request, require cooperation between client and server and support from the ORB.

List of Figures

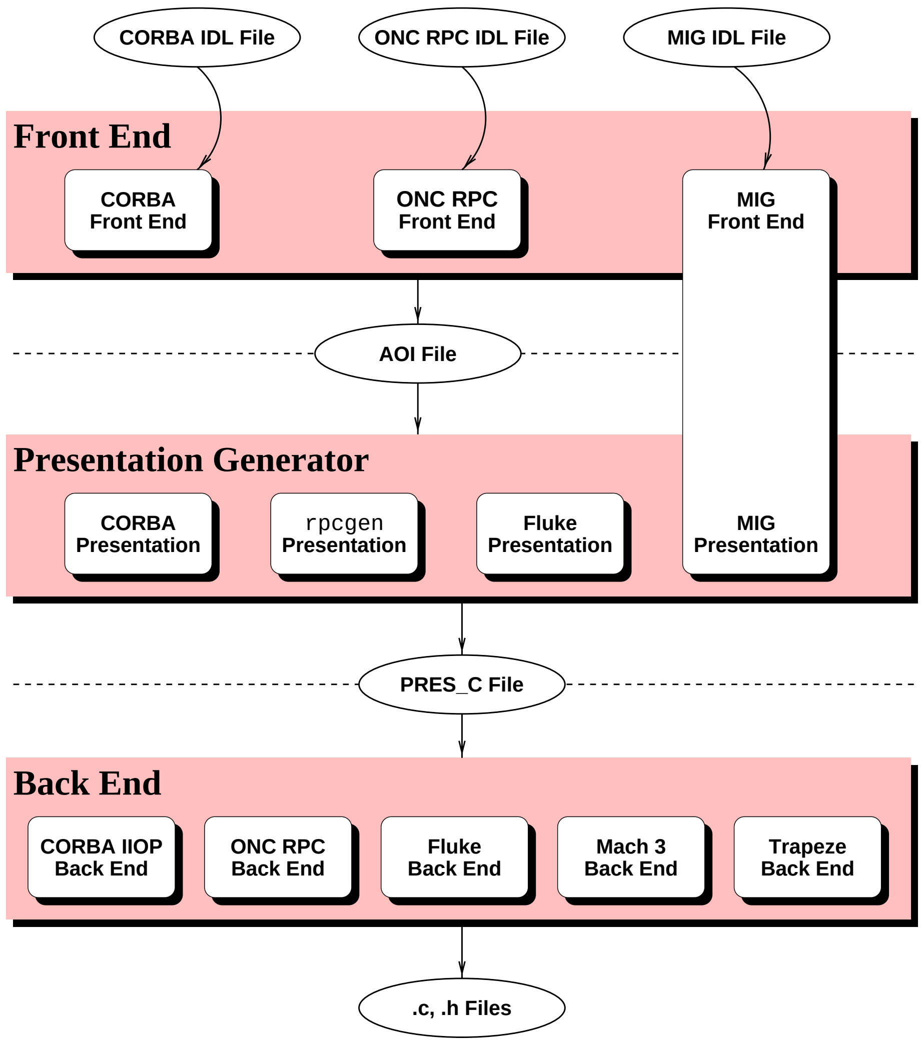

Figure 1: Overview of the Flick IDL compiler. Flick divides IDL compilation into three stages that communicate through intermediate files. As represented by the shaded boxes, each compilation stage is implemented primarily by a library of code that provides common functions and abstractions. Each particular front end, presentation generator, and back end is created by linking a relatively small amount of specialized code with the appropriate base library. The MIG front end and presentation generator are conjoined due to the nature of the MIG IDL [9].

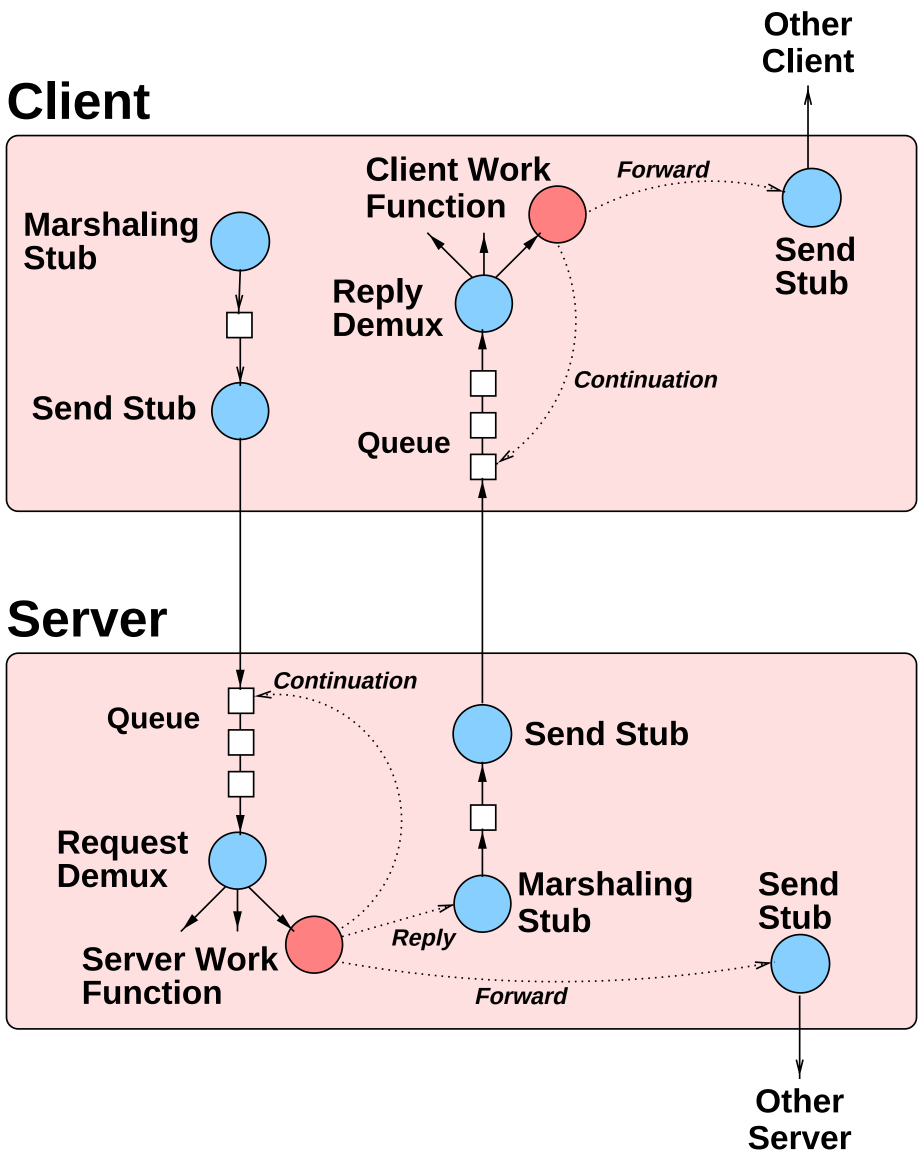

Figure 2: Overview of client/server communication through "decomposed" stubs. Unlike the standard (synchronous RPC) stubs produced by a normal CORBA IDL compiler, Flick's decomposed stubs allow for asynchronous requests and replies, efficient message forwarding, reuse of marshaled messages, and saving intermediate results.

List of Tables

Marshaling	*Interface*_Request *Interface*_*operation*_encode_request(
Stubs	*in and inout parameters*,
	CORBA_Environment *env);
	*Interface*_Reply *Interface*_*operation*_encode_reply(
	return and inout and out parameters,
	CORBA_Environment *env);
	*Interface*_Reply *Interface*_*operation*_encode_exception(
	CORBA_Environment *reply_env,
	CORBA_Environment *env);
Unmarshaling	void *Interface*_*operation*_decode_request(
Stubs	*Interface*_Request msg,
	in and inout parameters,
	CORBA_Environment *env);
	void *Interface*_*operation*_decode_reply(
	*Interface*_Reply msg,
	return and inout and out parameters,
	CORBA_Environment *env);
Send	void *Interface*_send_request(
Stubs	*Interface* target, *Interface*_Request msg,
	Invocation_id inv_id, Client client,
	CORBA_Environment *env);
	void *Interface*_send_reply(
	Client client, *Interface*_Reply msg,
	Invocation_id inv_id, *Interface* target,
	CORBA_Environment *env);
Work	void *Interface*_*operation*_do_request(
Functions	*Interface* target, *Interface*_Request msg,
	Invocation_id inv_id, Client client);
	void *Interface*_*operation*_do_reply(
	Client client, *Interface*_Reply msg,
	Invocation_id inv_id, *Interface* target);
Continuation	typedef void *Interface*_Request_Continuer(...);
Stubs	typedef void *Interface*_Reply_Continuer(...);
	void *Interface*_*operation*_continue_request(
	Interface target, *Interface*_Request msg,
	Invocation_id inv_id, Client client,
	*Interface*_Request_Continuer func, void *data);
	void *Interface*_*operation*_continue_reply(
	Client client, *Interface*_Reply msg,
	Invocation_id inv_id, *Interface* target,
	*Interface*_Reply_Continuer func, void *data);

Table 1: Summary of the stubs and functions that are part of Flick's "decomposed" CORBA presentation style. The italicized elements (e.g., *Interface*) are replaced with the appropriate names and elements from the input IDL file.