This paper should be referenced as:

Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Morrison, R., Munro, D.S. & Scheuerl, S. "Flask: An Architecture Supporting Concurrent Distributed Persistent Applications". University of St Andrews Technical Report CS/97/4 (1997).

# Flask: An Architecture Supporting Concurrent Distributed Persistent Applications

Graham N.C. Kirby, Richard C.H. Connor*, Quintin I. Cutts*,
Ron Morrison, Dave S. Munro and Stephan Scheuerl

School of Mathematical and Computational Sciences, University of St Andrews, St Andrews KY16 9SS
*Department of Computing Science, University of Glasgow, Glasgow G12 8QQ

## Abstract

Distributed application systems have become a popular and provenly viable computing paradigm. There are a number of reasons for this such as: the geographical dispersal of information; the improved reliability of multiple computer systems; and the possibility of concurrent execution of applications. As yet no single model of distribution has been pervasive and since the impact of failure semantics varies with the software architecture of applications, it is unlikely that one model will ever dominate. It is difficult to assess or even to compare the attributes of different models especially when run over the same data. This is often made more difficult in that most implementations of distributed models are closed systems with built-in protocols, failure reporting and concurrency control. The Flask architecture, presented here, takes the approach of providing a layered architecture which has the flexibility to support different models of distribution that can run over the same data. To demonstrate the feasibility of Flask an example distributed application is described using the architecture.

**Keywords**: distribution, persistence, concurrency, flexibility

# 1 Introduction

Distributed application systems have become a popular and provenly viable computing paradigm. There are a number of reasons for this such as: the geographical dispersal of information; the improved reliability of multiple computer systems; and the possibility of concurrent execution of applications. As yet no single model of distribution has been pervasive and since the impact of failure semantics varies with the software architecture of applications, it is unlikely that one model will ever dominate.

Distribution may be provided at many levels in a computer system. Some designs attempt to hide the distribution and create the illusion of a single system, while in contrast many programming languages provide language features that enable the programmer to exploit distribution. Many different models of distribution in persistent and object-oriented systems have been proposed and implemented [Lis84, Li86, Mos89, Wai89, HRK91] . Each of these models operates on the data in different ways, such as presenting different levels of transparency abstraction, utilising alternative concurrency control schemes and employing different fault-tolerance strategies. It is difficult to assess or even to compare the attributes of these models especially when run over the same data. This is often made more difficult in that most implementations of distributed models are closed systems with built-in protocols, failure reporting and concurrency control. The Flask architecture, presented here, takes the approach of providing a layered architecture which has the flexibility to support different models of distribution that can run over the same data.

Flask [MCM+94] was designed to provide an architecture that could enable different concurrency control schemes to be specified and run over persistent data. The key factor in the success of Flask is that as little as possible is built-in to the lower architectural layers, thereby providing increased functionality moving up the levels of abstraction. This provides the necessary flexibility but may also permit efficiency gains since many optimisations are achievable at a higher-level. The approach is comparable with that taken in RISC architecture where a simpler, more understandable hardware interface can be exploited more readily by software construction to forge efficiency gains. This design criterion has been used to extend Flask to accommodate models of distribution without compromising its flexibility. Indeed the revised architecture can be seen as more general, allowing different concurrency control models and different distribution models to be established without the need to re-implement entire systems. This paper outlines the Flask architecture and details how the architecture can be used to support distribution models.

As a proof of concept of Flask an example distributed application is described using the architecture. This consists of a distributed database with front-ends, query servers and back-ends running on different nodes. Section

3 describes the application, the particular choices made in implementing each of the layers, and how the architecture allows these implementation choices to be altered if required.

# 2 Overview of Flask

Flask is a layered architecture which has the flexibility to support different models of distribution and concurrency over the same data. The architecture eschews any fixed notion of concurrency control or any built-in model of distribution. Instead it provides a framework in which models can be defined and supported. One of the major difficulties in engineering such a system lies in the building of generic mechanisms to provide the facilities of data visibility restriction, stability, and atomicity, independently of the combination of these employed by a particular scheme.

The framework of the Flask architecture is shown in Figure 2 as a "V-shaped" layered architecture to signify the minimal functionality built-in at the lower layers. At the top layer the specifications of the model are used to guide the algorithms used to enforce them and can take advantage of the semantics of these algorithms to exploit potential concurrency. For example a particular specification may translate into an optimistic algorithm or alternatively a pessimistic one while the information they operate over remains the same. More importantly such an approach can accommodate different models of concurrency control and distribution.

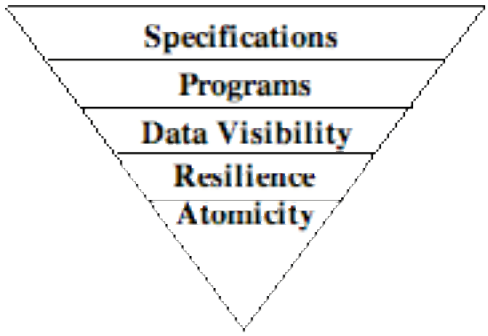


**Figure 2: V-shaped Layered Architecture**

The specification for each particular model identifies the events that are significant to that model. For example an atomic transaction scheme may highlight *start transaction*, *abort*, *commit* and *read* and *write* as the events of interest whilst a two-phase commit protocol may identify co-ordinator and participant *prepare* and *complete* events as the significant events. The programs which implement the specifications pass control information down the layers and expect to receive feedback information about these significant events from the lower layers. This enables the high-level definition, determination and handling of normal and error processing such as conflict detection or partial failure without the need for low-level built-in controls. It also frees the lower layers from the complexity of interference management and global fault-tolerance, and promotes considerable flexibility allowing different designs and implementations of memory management and recovery schemes at each node of a distributed system.

In Flask, the visibility of data is expressed in terms of the control of movement between a globally visible database and conceptual stores called access sets. Each action is associated with a local access set and may use other shared access sets. The interfaces to the data visibility are:

- Start action — *a cohesive set of operations on objects*
- End action
- Create and delete local access set — *private to an action*
- Create and delete shared access set — *shared among actions*
- Copy from access set to access set
- Remove object from access set
- Meld — *consistent merge with global state*

At the lowest level the atomicity layer ensures consistent update to the access sets and the global database. The failure resilience layer utilises this atomicity to effect an action *meld*. The term *meld* is used to describe the action of making updates permanent rather than terms like *commit* or *stabilise* since they imply specific meanings in particular models.

With distribution the mapping of an instantiation of a model onto a physical system is totally dependent on the model itself. The specification layer is in some sense global in that it administers the cohesion of the global

database. The specifications are translated into distributed algorithms which control the data visibility. The key idea here is that the significant events defined by a particular distribution scheme such as time-out, failure detection, duplication etc., are reported to the higher layers enabling the maintenance of the global cohesion. The manner in which the programs which drive the specifications are mapped onto physical nodes is dependent on the particular model. Similarly distributed data visibility is model-dependent. The global database is a model-defined combination of data local to each node whilst the scope of local and shared access sets may span more than one physical node. The resilience layer involves the maintenance of recovery mechanisms that may be required to work co-operatively or independently.

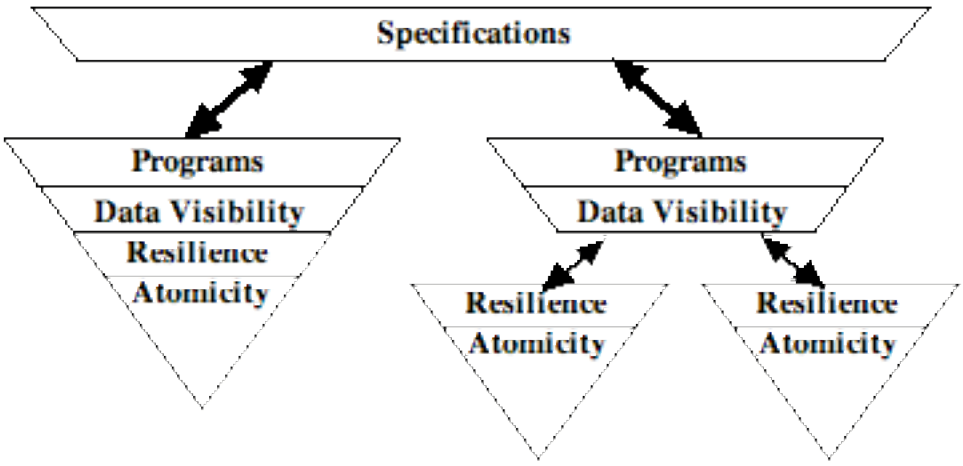


**Figure 3: Example of a Distributed Flask Model**

Figure 3 gives a pictorial representation of how Flask may be instantiated for an example distribution model. The specification of the model translates into a set of algorithms that is distributed across two *virtual* nodes. In one of these virtual nodes all the Flask layers are constrained to that node whilst the other virtual node shows shared data visibility supporting two separate resilience layers. It is important to realise that there is no fixed mapping between virtual and physical nodes in a real system.

Two systems which meet the requirements of the Flask architecture and which are major influences on the Flask approach are Stemple and Morrison's CACS system [SM92] and Krablin's CPS-algol system [Kra85] . The CACS system provides a framework in which concurrency control schemes can be specified. CACS is a generic utility for providing concurrency control for applications. The system does not actually manipulate any objects, but instead maintains information about their pattern of usage. In order for the system to collate this information, the attached applications must issue signals regarding their intended use of their objects. In return the CACS system replies indicating whether or not the operation would violate the concurrency rules. CPS-algol is an extension to the standard PS-algol system [PS88] that includes language constructs to support and manage concurrent processes. The concurrency model is essentially co-operative with procedures executing as separate threads and synchronising through conditional critical regions. Krablin showed that with these primitives and the higher-order functions of PS-algol, a range of concurrency abstractions could be constructed including atomic and nested transactions as well as more co-operative models.

# 3 Case Study: A Distributed Database

This section describes a simple distributed persistent application, a specialised database, which was developed to demonstrate the feasibility of the Flask architecture, using Napier88 [MBC+94] . The application consists of a database query server which runs on one node, accepting queries from database front-ends running on other nodes, and returning results from a database distributed over a number of nodes. This structure is illustrated in Figure 4:

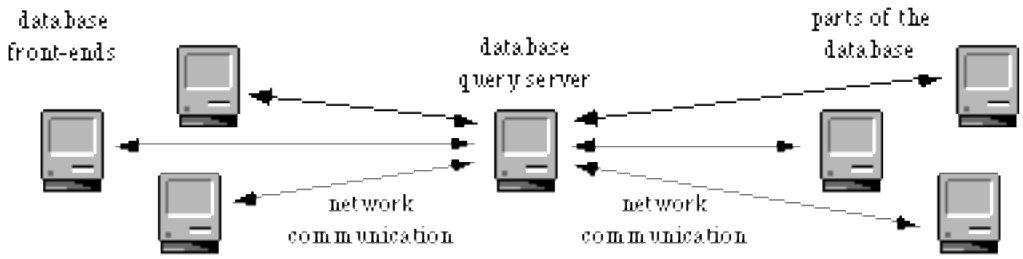


**Figure 4: The Application Structure**

The key feature is that the software running on each node is structured in the V-layered style, with peer-peer communication and concurrency coded in the higher layers. Building the application involved making a particular set of choices for the layers on each node. Section 4.4 describes what different choices could be made and the processes involved in changing them.

## 3.1 Application Details

The example application displays information about golf courses in Scotland and allows the user to book a series of rounds of golf, each during a particular time slot on a particular course. The main interface presents a map of Scotland and the positions of a number of golf courses, each represented by a flag icon. A number of interface widgets allow the user to specify constraints on the courses. The courses satisfying the current constraints are denoted by green flags, the others by red flags.

It should be emphasised that the purpose of developing the application was to demonstrate the practicality of the V-layered approach to providing flexibility, rather than to represent the state of the art in either database implementation or user interface design. An example of the interface in use is shown in Figure 5:

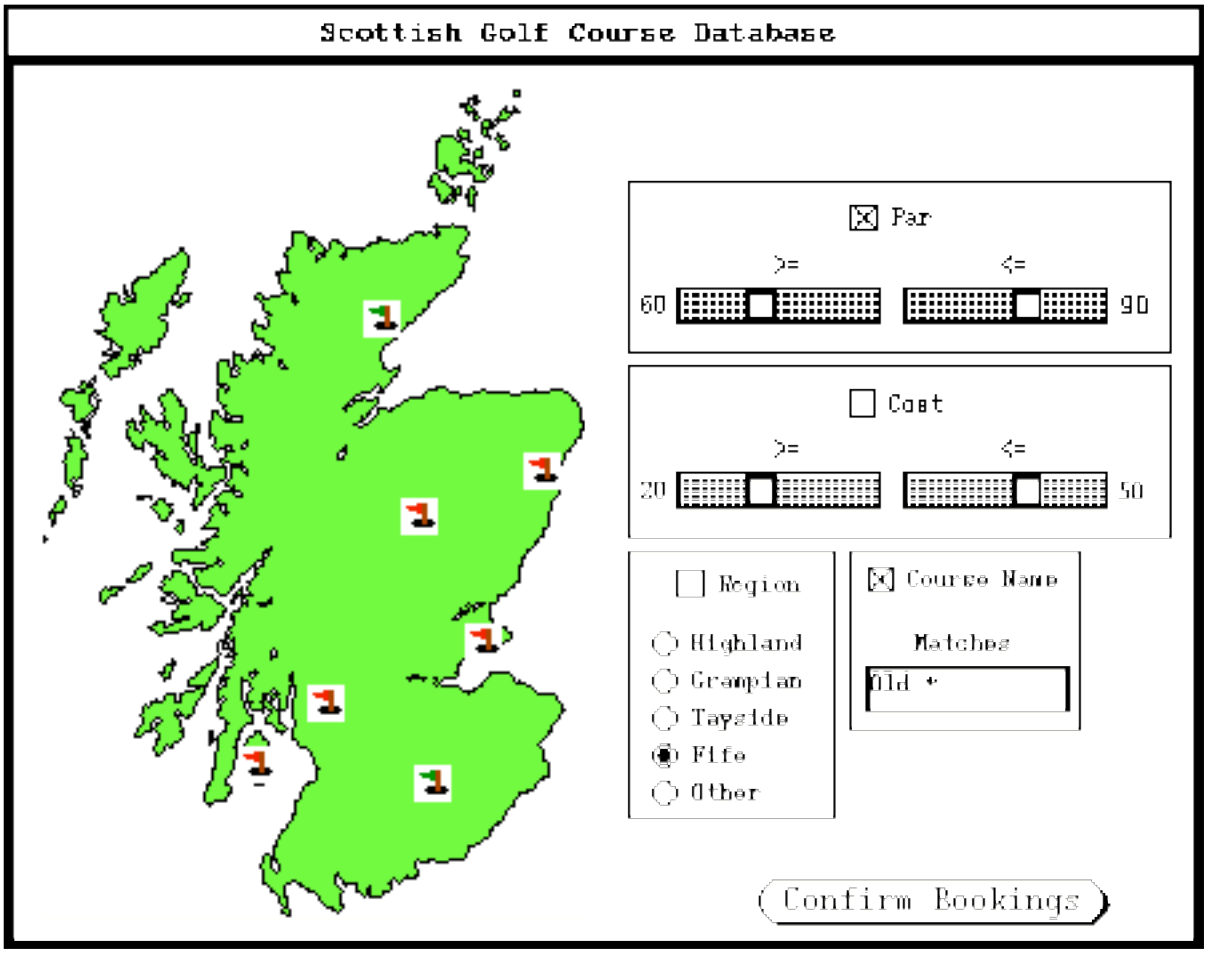


**Figure 5: The Starfield Golf Course Viewer**

This example shows a small number of golf courses for simplicity. Any of the constraints—par, cost per round, region, course name—may be selected individually by clicking on the appropriate check box. All the constraints which are selected are applied in succession.

Once a set of constraints has been applied to narrow the field of interest, the user may view the details of a particular course by double-clicking on the corresponding icon. This brings up a window showing a picture of the course, its par and region, and a dialogue which may be used to book a round on the course. An example is shown in Figure 6:


**Figure 6: Details of a Particular Golf Course**

The user may use the dialogue to select a time-slot on a particular day. Pressing the *Book Round* button results in a provisional booking for this time being attempted. The application checks that the proposed booking does not conflict with any other bookings, provisional or confirmed, already held in the database. If a conflict occurs, which is defined as the existence of another booking on the same course for a time within ten minutes of the attempted booking, the attempted booking is refused and all previous provisional bookings made by the user are also discarded. The user may then start the booking process again or quit the application. The application maintains a total of the number of golfers booked on the course for each day. This information may be accessed by the golf course administrator using a separate interface.

When the user has made a satisfactory set of provisional bookings they may be confirmed by pressing the *Confirm Bookings* button in the main viewer window. The bookings are then recorded in the database as permanent. If the user then proceeds to make an additional series of bookings, any unsuccessful attempt to make a provisional booking will result in only the provisional bookings made since the confirmation being discarded.

The purpose of the application as a demonstration should again be emphasised: the application behaviour described above was chosen to demonstrate a particular example of distribution and concurrency control.

## 3.2 Distribution and Concurrency Control Requirements

The previous section described the operation of the database application from the user perspective. Implementation of this description involves fulfilling a number of requirements regarding distribution and

concurrency control. To clarify terminology, Figure 7 shows the terms that will be used to refer to the various hardware node types and the application components which run on them. For simplicity the assumption is made that precisely one application component runs on each node. However, this assumption is not built in to the application or the architecture as a whole, and it could easily be relaxed.

| **hardware entity** | front-end node | query server node | database node |
|---|---|---|---|
| **software entity** | front-end controller | query server | database partition controller |
| **number present** | ≥ 1 | 1 | ≥ 1 |

**Figure 7: Hardware and Software Terminology**

The following requirements and options may be identified:

**Distribution**

**Coherency**: the data of interest here is the information about the golf courses and the bookings which have been made. This is stored on the database nodes. Since the query server is the only entity which communicates with the database partition controllers, any approach to data coherency can be taken so long as it is taken into account in the implementation of the query server. At one extreme a separate copy of the data could be held on each database node, with coherency maintained between them. At the other extreme the data could be partitioned so that any one data item was held on only one database node.

**Communication Level**: the data to be sent over the network between nodes includes spatial coordinates, text and photographic images. The requirement is for communication at a suitable level of abstraction, taking into account the probably conflicting criteria of ease of application implementation and efficiency of communication.

**Network Topology**: the nodes must be connected, at least logically, in a star configuration, with the query server node being the hub. Clearly with a larger system this would represent a bottle-neck and might need to be refined.

**Concurrency Control**

**User Level**: at the user level, when one user has made a provisional booking of a course, other users must be prevented from making conflicting bookings.

**Implementation Level**: at the implementation level, race conditions involving near-simultaneous conflicting booking attempts must be avoided.

## 3.3 Implementation

### 3.3.1 Napier88 and Persistence

The Napier88 system [MBC+94] is used as the basis for these experiments. Napier88 is a strongly-typed persistent programming language with a sophisticated type system, first-class procedures and environments. Napier88 is supported by a stable persistent object store through which all data is accessed. The use of Napier88 as a vehicle for this work has a number of advantages:

- The Napier88 system provides no explicit language constructs or store primitives for the expression of concurrency or transactions. The language thus has no preconceived view of concurrency that might complicate the provision of a range of models.
- Similarly, the system has no built-in model of distribution. Since all data in a Napier88 system is through the persistent store then the system is constrained by the bounds of the store.
- The Napier88 system is based on a layered architecture [BDM+90] that was specifically designed to support cost effective experimentation with persistent store design, concurrency, transactions and distribution in a persistent environment. The layered architecture provides an explicit layer for each of the many logical levels of architecture required by a persistent system.

The architectural layering has been chosen to take advantage of the persistence abstraction by ensuring that programs are not able to discover details of how objects are stored. This divides the architecture between the architectural layers that provide the persistent object store and those facilities that may be programmed by a supported programming language. Thus, a data format can be altered by the compiler without the need to alter the persistent store. The design of this layered architecture and an initial single-user implementation on Unix was produced by Brown [BR90] .

Although the discussion thus far has not strongly emphasised the persistence aspect of the architecture, the presence of orthogonal persistence is fundamentally important to the implementation of the layers above the persistent object store. For example:

- the language implementation may take advantage of the persistent store to improve efficiency in a number of ways [CCK+94, MBC+94] ;
- the user interface implementation may make use of persistent graphics facilities to provide a programming interface which is consistent with the manipulation of other types of data.

### 3.3.2 The Layering Structure

As described earlier, each persistent application in the Flask architecture is structured into a number of layers, with as much functionality as possible concentrated in the higher layers. Figure 8 shows the layers which may be identified, together with some of the particular layer instances which have been implemented in Napier88. These instances represent the choices available to the application implementor. They are described in more detail in the following sections. The stars in the third column indicate the instances which have been used in the database application.

| layer | instances | used in example |
|---|---|---|
| 0: hardware platform | Sun SPARC<br>DEC Alpha | *<br>* |
| 1: object store | shadow paged<br>DataSafe | *<br>* |
| 2: abstract machine/language | PAM/Napier88<br>PamCase/Napier88 | *<br>* |
| 3: peer-peer communication | raw sockets<br>Stacos object based type safe RPC | * |
| 4: concurrency control | atomic transactions<br>sagas | <br>* |

**Figure 8: Application Layers and Instances**

Using the Flask architecture for constructing a distributed persistent application involves choosing instances for each layer.

### 3.3.3 Persistent Object Stores Layer

The persistent object store implementations available include a concurrent after-look shadow paged store [Mun93] and the DataSafe store [SCM+96] . Both object stores run on multiple hardware platforms. There are a number of different mechanisms, such as write-ahead logging or shadow paging, which can provide the necessary resilience to support stable virtual memory.

Munro's store uses after-look shadow paging to provide stability. In a shadow paging system a page replacement algorithm controls the movement of pages between volatile and non-volatile store such that recovery will always produce a consistent state. To implement this a disk page table is used to maintain the correspondence between the virtual pages of the database and blocks on non-volatile store. After-image shadow paging writes each updated page to a free block and updates the disk page table to reflect the new mapping. The mechanism maintains a mirrored root block from which the last consistent map can be found. On a meld the new mappings, in addition to updated data pages, are written to non-volatile store and then the oldest root block is updated

atomically. Since after-image shadow paging always writes pages to new disk blocks, the original clustering of the blocks may be lost.

The DataSafe is a page log based mechanism, based on the DB cache [EB84] , which ensures the recoverability of a persistent store to a consistent state after system failure. Reads and writes are performed on faulted versions of persistent store pages held in volatile storage. The mechanism avoids writing non-melded pages of data back to the persistent store and thus avoids maintaining redundant undo information. When a meld is initiated, to atomically propagate updated pages to the persistent store, the updated pages are written atomically to a circular log called the safe. The safe is circular to enable all writes to the safe to be performed sequentially, and to bound the amount of data required for recovery. In the case of a successful meld these melded pages either remain in volatile storage to be reused or are written to the persistent store opportunistically: the safe ensures the recoverability of melded pages which have not yet been written to the persistent store. Should the safe become full during a meld a checkpoint propagates sufficient safe pages required for recovery to the persistent store to permit the meld to complete. During recovery safe pages required for recovery are read from the safe into volatile storage or the persistent store and normal processing resumes. During normal processing the latest version of every persistent store page is either in volatile storage or in the persistent store thus ensuring all page faults operate on the persistent store.

### 3.3.4 Abstract Machines and Languages Layer

The persistent abstract machine implementations available are the PAM [CBC+90] and PamCase machines. User programs are compiled down to the instruction set of the appropriate abstract machine.

The Persistent Abstract Machine is built on a heap-based architecture that is designed as a convenient way of supporting the block retention needed for the use of first-class procedures and is responsible for implementing the necessary primitives to support polymorphism and abstract data types. Since the abstract machine does not allow direct access to the persistent store it ensures that the compilation system is unaware of the implementation of object storage, thus separating the use of an object from the way it is stored. A design aim of the architecture provides the persistent object store as the only available storage for the abstract machine. This means that there is only one storage mechanism and one possible way of exhausting it.

The PamCase machine is derived from the PAM with the primary aims of improving execution speed and reducing object management costs. This is achieved using a different first-class procedure implementation, based on that of the CASE machine [DM88] . An efficient implementation of first-class procedure values is essential since they are used to support the inclusion of segmented code objects within the persistent store. Such code values are used very heavily within the persistent system as the basis of protection mechanisms, application construction architectures, concurrency and distribution protocols.

The PamCase interface to the persistent object store is identical to that of PAM. The interface to the language implementation differs since the instruction set is different, thus a different compiler is used with PamCase.

### 3.3.5 Communication Protocols Layer

The communications layer implementations available are at the raw socket level, the Stacos model [Mun93] , and type-safe RPC [Mir95] . The programmer may develop new protocols as described in Section 3.

Communication at the socket level involves the reading and writing of byte streams. The application programmer may interpret the byte streams as required. The Stacos model was mentioned in Section 2.2: it allows programs running in one persistent store to scan the contents of other persistent stores and to make deep copies of objects in other stores.

The type-safe RPC mechanism developed by Mira da Silva [Mir95] allows a program running in one persistent store to invoke a procedure in another store and to have either a time-out message or a result, as a copy of a persistent object in the remote store, returned to the calling store. This mechanism is used in the example application.

### 3.3.6 Concurrency Control Protocols Layer

The concurrency control protocols appropriate to the particular application may be defined by the programmer using the Flask model. Protocols which have been defined in this way include atomic transactions [Dav73] and sagas [GS87] , which are used in the example application.

The saga model is an attempt to increase concurrency in systems which execute long-lived transactions. The transactions are broken down into a number of short independent atomic transactions which are organised into a

saga. The completion of a saga depends on the success of the serial execution of every component transaction. The failure of a saga, caused either by system crash or by the failure of a component transaction, requires that compensating transactions are executed to compensate for the globally visible effects of the component transactions. Figure 9 illustrates the structure of a saga. A long-lived transaction has been broken down into a number of atomic transactions $\{T_1, T_2, \ldots, T_n\}$, the first n-1 of which are associated with compensating transactions $\{C_1, T_2, \ldots, C_{n-1}\}$.

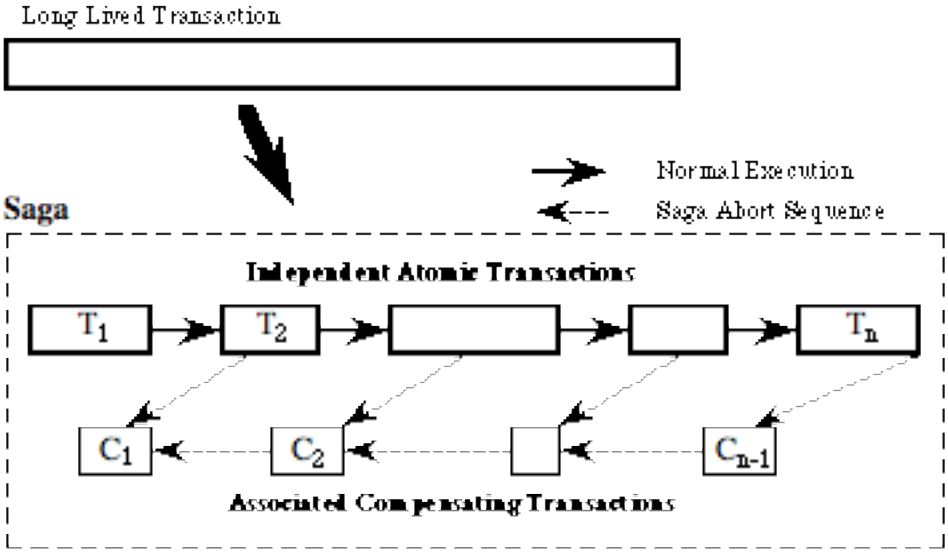


**Figure 9: Structure of a Saga**

If a component transaction $T_k$ fails, the saga either retries that component or executes the sequence of compensating transactions $\{C_{k-1}, C_{k-2}, \ldots, C_1\}$. The mechanism is restricted in that the programmer must be able to break the long-lived transaction down into a number of components for which compensating transactions must be constructed. Transaction $T_n$ needs no compensating transaction since it is an atomic transaction whose failure leaves no effects to be compensated for.

To implement sagas in the Flask framework it is necessary to define the significant control events. These are:

- begin saga — *start a saga*
- end saga — *terminate a saga successfully*
- begin transaction — *start a transaction within current saga*
- end transaction — *terminate current transaction and commit updates*
- abort transaction — *abort current transaction and retry*
- abort saga — *abort current transaction and entire saga*
- compensate — *execute compensating transaction for a given transaction*
- read — *read an item in the local access set*
- write — *update an item in the local access set*

Once the saga model has been defined, it is the task of the application programmer to embed the appropriate control events within the application code. The programmer must also define the compensating transactions which compensate for the committed changes of an aborted saga. For example an *end saga* event occurs when the *Confirm Bookings* button is pressed, an *end transaction* event occurs when a provisional booking is made successfully, and an *abort saga* event occurs when an attempt to make a provisional booking conflicts with an existing booking. The compensating transaction for a given transaction subtracts the number of golfers booked from the daily total for the course. Note that it does not simply restore the total to the value as it stood at the start of the transaction, since other transactions may have updated the total in the interval between the end of the transaction and the abort of the saga.

An update in the saga model is either local or globally visible. Thus only local access sets are used in the Flask specification; shared access sets are not required. Since in the example application sagas are implemented above an RPC communication layer, it is also necessary to define the control events for the RPC protocol. These are:

- register service — *register a service on local node*
- invoke service — *call a service on a given node*

- receive service request — *receive a request for a service on local node*
- timeout — *timeout when reply not received after given interval*
- send reply — *send a reply to a service request*
- receive reply — *receive a reply from a service*

The Flask specifications of the sagas and the RPC protocol are interlinked: for example, when an RPC timeout occurs the current transaction could retry the remote procedure call or it could abort. For simplicity the latter option is implemented in the example application, thus the occurrence of a *timeout* event in the RPC specification leads to an *abort saga* event in the saga specification.

### 3.4 Application Reconfiguration

This section has described some of the choices currently available to the implementor of a distributed persistent application using the Flask architecture, and one particular set of choices made in implementing the example database application. It would be possible to reconfigure the application by altering choices at various layers. The amount of work involved varies with the different layers.

- A node's implementation of layers 1 or 2 could be easily replaced by another, or run on a different supported hardware platform, since all implementations conform to common interfaces. A new implementation of a layer involves a significant effort, and is guided by the standard interfaces with the layers above and below.

- An implementation of layer 3 for a particular pair of nodes could be replaced by another, by rewriting the part of the application which deals with peer-peer communication and the coherency model. Implementation of a new distribution layer involves defining the significant events and using the Flask framework to develop the control code to run on each node. Currently this is done by hand-coding but it is a research goal to automate the process from the event definitions.

- The same applies to the replacement of the concurrency control layer.

## 4 Conclusions

The popularity of distributed application systems as a useful computing paradigm has led to construction of numerous and varied models. The conventional approach to constructing these systems is to build them from a bottom-up hierarchy of primitives which govern the models behaviour. This method, usually followed for performance reasons, makes it difficult to assess or even to compare models and often necessitates expensive reconstruction costs.

Flask is a generic layered architecture which is designed to provide sufficient flexibility to enable it to support many models. The Flask V-shaped structure builds in as little as possible at the lower layers, not only to provide the required flexibility, but also to free the lower layers from the burden of interference management and failure control. This greatly simplifies the construction requirements of these bottom layers and permits a variety of implementations to co-exist and be interchanged without the need for complete system rebuilds. Flask can thus be seen as a good platform for building, testing and comparing models of distribution and concurrency.

To demonstrate the feasibility and flexibility of this approach a persistent distributed application developed under the Flask architecture was described. The components of the instantiation involve three different hardware platforms, two object stores, two abstract machines, one communication and coherency protocol, and one concurrency control protocol. running over four nodes.

## 5 Acknowledgements

CACS was developed by David Stemple and Ron Morrison, with Chris Barter, Jason Denton, Gary Kirkpatrick and Barry Pretsell. The work was supported by ESPRIT III BRA 6309 — FIDE$_2$ and EPSRC Grant GR/J67611. Richard Connor is supported by EPSRC Advanced Fellowship B/94/AF/1921.

## 6 References

[BDM+90] Brown, A.L., Dearle, A., Morrison, R., Munro, D. & Rosenberg, J. "A Layered Persistent Architecture for Napier88". In **Security and Persistence**, Rosenberg, J. & Keedy, J.L. (ed), Springer-Verlag, Proc. International Workshop on Security and Persistence, Bremen, 1990 (1990) pp 155-172.

[BR90] Brown, A.L. & Rosenberg, J. “Persistent Object Stores: An Implementation Technique”. In **Implementing Persistent Object Bases, Principles and Practice**, Dearle, A., Shaw, G.M. & Zdonik, S.B. (ed), Morgan Kaufmann, Proc. 4th International Workshop on Persistent Object Systems, Martha's Vineyard, USA (1990) pp 199-212.

[CBC+90] Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A. & Morrison, R. “The Persistent Abstract Machine”. In **Persistent Object Systems**, Rosenberg, J. & Koch, D.M. (ed), Springer-Verlag, Proc. 3rd International Workshop on Persistent Object Systems, Newcastle, Australia (1990) pp 353-366.

[CCK+94] Cutts, Q.I., Connor, R.C.H., Kirby, G.N.C. & Morrison, R. “An Execution Driven Approach to Code Optimisation”. In Proc. 17th Australasian Computer Science Conference, Christchurch, New Zealand (1994) pp 83-92.

[Dav73] Davies, C.T. “Recovery Semantics for a DB/DC System”. In Proc. ACM Annual Conference (1973) pp 136-141.

[DM88] Davie, A.J.T. & McNally, D.J. “CASE - A Lazy Version of an SECD Machine in a Flat Environment”. University of St Andrews Technical Report Staple/StA/88/2 (1988).

[EB84] Elhardt, K. & Bayer, R. “A Database Cache for High Performance and Fast Restart in Database Systems”. ACM Transactions on Database Systems 9, 4 (1984) pp 503-525.

[GS87] Garcia-Molina, H. & Salem, K. “Sagas”. In Proc. ACM SIGMOD International Conference on Management of Data (1987) pp 249-259.

[HRK91] Henskens, F.A., Rosenberg, J. & Keedy, J.L. “A Capability-Based Distributed Shared Memory”. In Proc. 14th Australian Computer Science Conference (1991) pp 29.1-29.12.

[Kra85] Krablin, G.L. “Building Flexible Multilevel Transactions in a Distributed Persistent Environment”. In Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland (1987) pp 86-117.

[Li86] Li, K. “Shared Virtual Memory on Loosely Coupled Multiprocessors”. Ph.D. Thesis, Yale University (1986).

[Lis84] Liskov, B. “Refinement - From Specification to Implementation the Argus Language and System”. In Proc. Advanced Course on Distributed Systems Methods and Tools for Specification, University of Munich (1984).

[MBC+94] Morrison, R., Baker, C., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Munro, D. “Delivering the Benefits of Persistence to System Construction and Execution”. In Proc. 17th Australasian Computer Science Conference, Christchurch, New Zealand (1994) pp 711-719.

[MBC+94] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. “The Napier88 Reference Manual (Release 2.0)”. University of St Andrews Technical Report CS/94/8 (1994).

[MCM+94] Munro, D.S., Connor, R.C.H., Morrison, R., Scheuerl, S. & Stemple, D. “Concurrent Shadow Paging in the Flask Architecture”. In **Persistent Object Systems, Tarascon 1994**, Atkinson, M.P., Maier, D. & Benzaken, V. (ed), Springer-Verlag, Proc. 6th International Workshop on Persistent Object Systems, Tarascon, France (1994) pp 16-42.

[Mir95] Mira da Silva, M.M. “Automating Type-safe RPC”. In Proc. 5th International Workshop on Research Issues on Data Engineering: Distributed Object Management, Taipei, Taiwan (1995) pp 100-107.

[Mos89] Moss, J.E.B. “Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach”. In **Database Programming Languages**, Hull, R., Morrison, R. & Stemple, D. (ed), Morgan Kaufmann, Proc. 2nd International Workshop on Database Programming Languages, Salishan Lodge, Gleneden Beach, Oregon (1989) pp 358-374.

[Mun93] Munro, D.S. “On the Integration of Concurrency, Distribution and Persistence”. Ph.D. Thesis, University of St Andrews (1993).

[PS88] PS-algol "PS-algol Reference Manual, 4th edition". Universities of Glasgow and St Andrews Technical Report PPRR-12-88 (1988).

[SCM+96] Scheuerl, S.J.G., Connor, R.C.H., Morrison, R. & Munro, D.S. "The DataSafe Failure Recovery Mechanism in the Flask Architecture". To Appear: 19th Australian Computer Science Conference, Melbourne, Australia (1996).

[SM92] Stemple, D. & Morrison, R. "Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach". In Proc. 15th Australian Computer Science Conference, Hobart, Tasmania (1992) pp 873-891.

[Wai89] Wai, F. "Distributed Concurrent Persistent Languages: an Experimental Design and Implementation". Universities of Glasgow and St Andrews Technical Report PPRR-76-89 (1989).