Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocol

Florian Matthes, Rainer Müller, Joachim W. Schmidt
Universität Hamburg
Vogt-Kölln Straße 30
D-22527 Hamburg, Germany
matthes,mueller,J_Schmidt@informatik.uni-hamburg.de

Abstract

The Tycoon Store Protocol (TSP) specifies a clean interface between the frontend and the backend of fully integrated persistent environments. In contrast to high-level relational or object-oriented database languages, TSP is based on a low level, untyped, but highly flexible tagged store model that is particularly well-suited for the implementation of higher-order persistent polymorphic languages. We describe the TSP operations in some detail and give insight into TSP's design rationale. We also report on the existing set of TSP-compliant backends including TSP adaptors to commercial object stores. A TSP client can choose dynamically between these backends and it is possible to exchange complex object graphs between all TSP-compliant stores via a platform-independent external data representation.

1 Introduction and Motivation

Virtually all systems that have to work with large-scale persistent data have a system architecture where there is a clean separation between a *frontend* performing data manipulation and visualization and a *backend* responsible for reliable persistent bulk data storage. Since actions in the frontend trigger data access operations performed by the backend, this separation naturally leads to a client-server architecture, where a *store protocol* defines possible interactions between a client program and a database server.

For standard database applications like business applications, SQL is the store protocol of choice: SQL provides *standardized* high-level, application-oriented store operations, it is available on a wide range of platforms, and there are standardized mechanisms to perform the frontend to backend binding (static language bindings via preprocessors, dynamic SQL invocation via an application programming interface like ODBC). The ODMG standard [Cat94a, Cat94b] and the ISO STEP standards [ISO92] define similar high-level store protocols to access object-oriented databases and repositories of construction data, respectively.

Unfortunately, these commercially well-supported store protocols turn out to be of little help in the construction of fully integrated persistent programming environments since their hard-wired high-level data models do not match the flexibility requirements of persistent languages (see [MMS92, Mül91] for a more detailed discussion):

- Full persistence abstraction [AB87] requires a uniform, efficient element-oriented access to persistent and non-persistent data, code and threads [MS95] which leads to an overly expensive two-level store management if implemented via standard database call interfaces.
- Polymorphic data structures, higher-order (polymorphic) functions and user-defined abstract data types are very hard to realize in monomorphic high-level data models.
- Incremental program construction and reflective programming techniques require schema modifications at runtime and dynamic binding techniques which are beyond the scope of most standard store protocols.

As a consequence of these difficulties, virtually each persistent programming environment comes with its own tailored object store (and vice versa). If one looks at existing pairs of object stores and their persistent languages, like Exodus and E, O_2 and O_2C or ObjectStore and Persistent C, their internal object store protocols are often well below the abstraction level of SQL or of the ODMG languages. Furthermore, these store protocols are characterized by heavy dependencies between language processor and storage subsystem and by several ad-hoc design decisions regarding object layout and object lifetime.

In this paper we describe the Tycoon Store Protocol (TSP) that has been developed during the last three years by our group to overcome the above problems and to provide a well-defined, standardized interface between the frontend and the backend of fully integrated persistent environments. This interface is based on the widely accepted notion of an *untyped persistent heap* ([MS88, BM91, KKD89]). Despite the fact that TSP has been implemented and evaluated in the context of the Tycoon project [MS93], TSP is fully independent of the Tycoon system and supports multiple (generic) clients and multiple database servers.

This paper is organized as follows: We first sketch how TSP contributes to the construction of scalable persistent systems. In section 3 we describe the untyped TSP store model followed by a detailed discussion of the TSP operations in section 4. Section 5 reports on our experience implementing TSP backends by adaptors to existing object stores and extending the TSP functionality systematically by TSP-compliant layers.

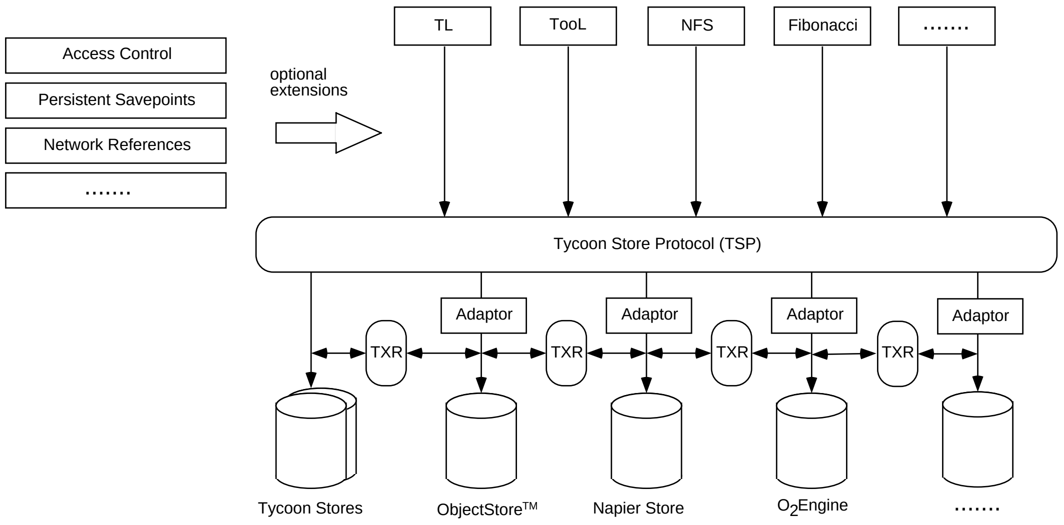

Figure 1: The role of the Tycoon Store Protocol in scalable persistent systems

2 On the Construction of Scalable Persistent Systems

TSP has been designed to support the construction of *scalable* persistent systems, i.e., configurable systems where the complexity and size of the entire system is proportional to the operational support required by a particular client. For example, it should be possible to add features like multi-user access, distribution and access control to the basic persistent storage services without perturbating the overall system architecture of a persistent system. On the other hand, the performance of clients that do not need these advanced features should not be deteriorated by the extensibility of the system.

Figure 1 explains the role of TSP in such a persistent system scenario. The data access operations defined in TSP are issued by client programs and executed by database servers. Currently, TSP is used by the runtime system of two persistent languages (TL [MS92], TooL [GM95]) developed at Hamburg University and by a network file system (NFS) implementation of our group that makes it possible to read and manipulate persistent store objects like ordinary Unix files or directories enhanced by a transactional recovery mechanism. Further TSP clients like runtime systems for other persistent languages such as Fibonacci [ABD+94] are under development. Each of these clients can choose dynamically between one of the following TSP implementations with distinct operational qualities and performance characteristics:

- *Tymem* is a portable main memory store developed by our group that achieves persistence and recovery through standard file system services.
- *Tysin* is a portable disk-based single user store that was developed by our group specifically for systems without virtual memory support like Microsoft Windows and Macintosh OS.
- *Napier* is a store adaptor to the Napier disk-based single user store for SunOS systems [Bro89]. This store makes heavy use of memory-mapped files [RHB+90] to achieve very fast object addressing.
- *Object* is a store adaptor to the commercial product ObjectStore [LLOW91], a multi-user store with a client-server architecture.

Furthermore, TSP is intended to be a stackable protocol. For example, it is possible to add access control, persistent savepoints or version management on top of arbitrary stores that adhere to TSP. These optional system layers utilize TSP as their lower and upper interfaces and extend the semantics of individual TSP operations (e.g., store object update) and provide additional TSP operations (e.g., rollback to savepoint).

TSP defines a stable application programming interface for client programs (expressed by a standard C header file) that is complemented by a platform-independent external data representation (TXR) that can be generated and parsed by all TSP backends. The TXR makes it possible to transfer arbitrary complex, graph-structured persistent store objects via a linear data representation from one backend to another without exposing private implementation details of individual persistent stores. We have found the TXR to be extremely valuable for backup, communication, migration and system evolution purposes. Some of these application areas are described in [MMS95a, MMS95b].

A client can use TSP to access multiple stores (possibly of multiple architectures) concurrently and a multi-user database server like ObjectStore supports multiple TSP clients at a time which may run on different client nodes in a local area network accessing a shared persistent store.

3 An Untyped Store Model

A primary design goal behind TSP has been to provide efficient data storage as independent as possible from the data and language model supported by TSP store clients. In particular, TSP should be capable of supporting polymorphically typed models where the components of store objects can contain values of multiple types that cannot be fixed in a separate schema definition phase. TSP therefore uses an *untyped* low-level store model, i.e., there is no separate dictionary of type or schema information maintained by the backend. Instead of this, store values are made self-descriptive by imposing a regular object layout and a uniform tagging scheme. As a result, TSP achieves data model independence without sacrificing the advantages of self-descriptive statically-typed databases.

3.1 Atomic Data Values

On the one hand, TSP is to be defined without knowledge of the particular base types (integer, string, real, ...) and type constructors (record, object, array, set, list, ...) supported by its various clients. On the other hand, TSP has to perform algorithms on store objects like concurrency control, garbage collection and portable data exchange that require a limited knowledge of the semantics of data values. Furthermore, TSP has to provide efficient access mechanisms to read and update atomic data values like integers, addresses or object identifiers.

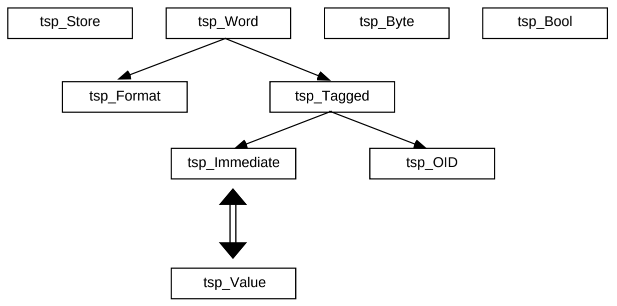

Figure 2: The type hierarchy underlying TSP

To match best the data modeling requirements of a given TSP client / server pair the TSP definition is *parameterized* by the following "abstract" atomic types (see figure 2):

tsp_Bool is a parameter type for TSP operations with the constants *tsp_TRUE* and *tsp_FALSE*.

tsp_Word is the supertype from which most other TSP types like object identifiers or tagged values are derived as subtypes. In particular, the size of *tsp_Word* values has to be greater or equal to the size of values of these other types. Typical bit sizes for tsp_Word values are 16, 32 or 64.

tsp_Byte is the type of values stored in byte array objects.

tsp_Value is the type of values manipulated by client programs.

Technically, these parametric type definitions are provided by a platform and storage-system dependent C header file (e.g. *tsp32.h*). This file has to precede the TSP C header file that is shared between all TSP implementations. The TSP header file defines additional derived or abstract types as shown in figure 2.

The type *tsp_Tagged* is the union of values from type *tsp_OID*, the type of persistent object identifiers, and of type *tsp_Immediate*, the type of TSP immediate values. The type *tsp_Tagged* is used in those contexts where both *tsp_OID* and *tsp_Immediate* values are valid. As the type name *tsp_Tagged* indicates, it is possible to distinguish OIDs from immediate values at runtime based on a tagging scheme.

The conversion between tagged store values (*tsp_Immediate*) and client values (*tsp_Value*) is encapsulated by two TSP mapping functions:

```
tsp_Immediate
   tsp_valueToImmediate(tsp_Store s, tsp_Value val);
tsp_Value
   tsp_immediateToValue(tsp_Store s, tsp_Immediate imm);
```

It should be noted that the size of client values is conceptually not dependent on the word size of the TSP implementation. Thus, the bitsize of a value of type *tsp_Value* can be smaller, equal or larger than the bitsize of a value of type *tsp_Word*. For example, in some stores polymorphic array objects have to implemented with an additional level of indirection such that client values can be larger than the slot size. The actual size supported by an TSP implementation can be retrieved by the TSP operation:

```
tsp_Word tsp_nValueBits(tsp_Store s);
```

Values of type *tsp_Format* are used to define the format of structured persistent objects in the store and are explained in the next section.

3.2 Structured Data Objects

TSP supports only two kinds of persistent data structures which are called array objects and byte array objects. In any context where both array and byte array objects are valid, we use the term object. Therefore, array objects and byte array objects are specializations of type object. Since there are no instances of type object, this type is similar to an abstract class in C++.

Every object has an object identifier (OID) that is guaranteed to be unique within a given TSP store. Object identifiers can be stored in arrays to establish bindings between objects. TSP does not guarantee object identifiers to be immutable over the lifetime of an object. For example, some TSP implementations re-assign OIDs during garbage collection. However, any implementation of the TSP guarantees that bindings between objects within the store are preserved.

Objects are allowed to hold atomic data values (as described in the previous section) only. Complex objects are constructed by establishing bindings between objects and not through object nesting. Since all TSP operations on objects have reference semantics, the signatures of TSP object operations uniformly use the type *tsp_OID*.

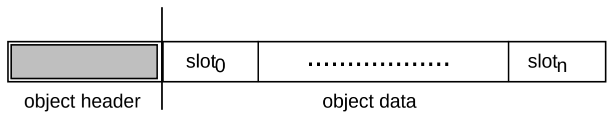

Figure 3: Structure of a TSP object

As depicted in figure 3, a TSP object conceptually consists of two parts: an object header (holding information about the object size, object mutability and object format) and a number of slots. When an object is created, the number of slots and the format have to be specified. An object is allowed to contain no slots at all. The first slot in an array has index 0. Conceptually, the maximum number of slots of a TSP object is infinite but there might be TSP implementations that define an upper limit. However, this limit is expected to be much larger than the page size of a traditional database system. In some persistent applications (e.g. B-Tree index management) TSP clients would like to create objects that fit exactly onto a single page of the underlying persistent store. In such applications, the function

tsp_Word tsp_nWordsPerPage(tsp_Store s);

can be used to return an integer value to be used as a size specification for a subsequent array object creation operation. Except for this "hint", page sizes of the TSP backend are fully transparent to TSP clients.

TSP provides operations to inspect the format and size of an object. The operation (*tsp_getFormat*) returns a value of type *tsp_Format* which is organized as a bit vector. The size of the bit vector depends on the TSP implementation. The lowest two bits of the vector are reserved for TSP purposes to distinguish between the three built-in array formats described below. The number of remaining format bits available to the client can be retrieved by the following TSP operation:

tsp_Word tsp_nFormatBits(tsp_Store s);

These bits can be utilized by TSP clients to define specialized array or byte array objects, for example arrays of real numbers, as subtypes of the following two predefined object types:

Array Objects: TSP supports two builtin formats for array objects. The format determines which values can be stored in the slots of an array object.

Array objects with format *tsp_Format_TAGGED* contain values of type *tsp_Tagged*. In particular, such array objects must not contain untagged values of type *tsp_Word* because a TSP server uses the tagging information to differentiate between object identifiers and immediate values.

Array objects with format *tsp_Format_WORD* contain untagged values of type *tsp_Word*. It is not allowed to store object identifiers in such arrays. On the other hand, the full bit size of type *tsp_Word* is available in each array slot. The format information *tsp_Format_WORD* is used heavily by the garbage collector and the TSP operations *tsp_intern* and *tsp_extern* to reduce the execution time for these operations.

Byte Array Objects: The size of an individual slot in a byte array is given by the size of values of type *tsp_Byte*. The predefined format code for byte arrays is *tsp_Format_BYTE*.

After creation, the slot values of an object can be updated. Therefore, the state of a newly created object is called mutable. It is possible to set the state of an object to immutable (*tsp_setImmutable*). Subsequently, it is not allowed to update the object and it is not possible to change the state of an object from immutable back to mutable. The TSP operation *tsp_isImmutable* can be used to test whether an object is immutable. The immutability information is exploited by many TSP implementations to optimize storage management, concurrency control and recovery.

4 Overview of the Tycoon Store Protocol

Due to the simplicity of TSP's data model, it is possible to define concisely all its operations which are clustered into operations on persistent stores, failure handling, that are detailed in the following sections.

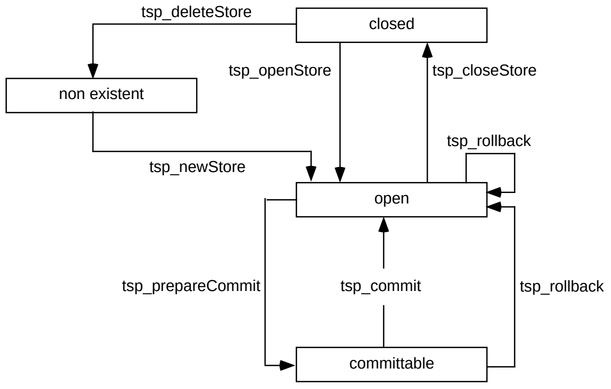

Figure 4: Store state diagram

4.1 Operations on Persistent Stores

Figure 4 shows the possible states of a TSP store and the operations that trigger state transitions of a store.

A TSP store must be created before it can be used. A TSP store can be created only within a TSP session. A client starts a session by calling the TSP function *tsp_init*. A session is finished by terminating the operating system process that started the session. It is not allowed to call *tsp_init* multiple times within a session.

A TSP store has a unique root value of type *tsp_Tagged* that serves as an entry point into the store. On creation of a store, an initial root value has to be specified. The initial root value must be an immediate value. If the store can be created it changes its state from *non_existent* to *open* and a handle for the store is returned to the client. The store handle is used by virtually all TSP operations to identify the store for which the operation is to be executed. This is necessary because TSP allows a client to operate on multiple stores simultaneously (some implementations, like the Napier Store, restrict the number of stores to one). Cross references between different stores are not supported by the basic TSP.

Consistent states of an object store can be made persistent ("committed"). The initial state of a newly created store is committed automatically. Subsequently, a client creates objects and links them together using TSP operations to form an object graph in the store. To make the object graph persistent, the client has to perform the following three steps:

1. Update the root of the store with the object identifier of the root of the graph (*tsp_setRoot(store, oid)*). This object is also called the root object of the store. Alternatively, if an object store already contains some objects, a newly created object graph can be made reachable through its OIDs from the existing object graph.

2. Ensure that the store can be committed (*tsp_prepareCommit)*. This function must be called before calling *tsp_commit* and returns *tsp_TRUE* to indicate that a subsequent *tsp_commit* will succeed (otherwise *tsp_FALSE* is returned but no automatic rollback is triggered). The state of the store changes from *open* to *committable*.

3. Define a new persistent state by committing the changes made to the store (*tsp_commit*). When a store is committed, all modifications to the store are written atomically to the persistent storage device the store resides on. Because TSP defines persistence by reachability, the root object and any object transitively reachable from the root object are made persistent. Moreover, the store changes its state from *committable* to *open*.

The description above assumes that a TSP backend supports an explicit two-phase commit protocol. This store property can be inquired at runtime with the TSP function *tsp_hasTwoPhaseCommit*. If no two-phase commit is supported by a store *tsp_prepareCommit* always returns *tsp_TRUE*.

As long as the store is in the *open* state, the client has the ability to create new objects, to modify objects and to commit the new state of the store. It is also possible for a client to rollback to the last committed state of the store, by calling *tsp_rollback*. *tsp_rollback* deletes all objects the client has created since the last call to *tsp_commit* and cancels the modifications of all other objects. This function can also be called if the state of the store is *committable*.

If an open store is no longer used within a session, it can be closed by calling *tsp_closeStore*. *tsp_closeStore* does not commit the store automatically. On successful completion of the operation, the store changes its state to *closed* and TSP invalidates the store handle. It is not allowed to call a TSP function with an invalidated store handle (the result of such operations is undefined). The only valid TSP operations that can be applied to a *closed* store are *tsp_openStore* and *tsp_deleteStore*. The first one must be called before working on a store and sets the state of the store to *open*. The second one, *tsp_deleteStore*, irrevocably destroys the store and sets its state to *non_existent*.

The following table summarizes the restrictions on the set of TSP operations that are applicable in a given state of the store.

Store State	Valid Operations
open	all TSP operations except *tsp_deleteStore*, *tsp_commit*
closed	*tsp_openStore*, *tsp_deleteStore*
committable	*tsp_commit*, *tsp_rollback*

4.2 Failure Handling and Recovery

If any of the TSP operation cannot be executed correctly, a standard failure handling procedure is called that prints an error message and then terminates the client process undoing all changes to the store since the last commit operation. TSP provides operations to substitute this default failure handling procedure and to retrieve the currently defined failure handler. The client-defined handler must be of type *tsp_FailureHandler*. A handler has two parameters. The type of the first one is *tsp_Store* (a store handle) and the type of the second one is *tsp_Failure* (the reason of the failure). The client process is also terminated if a failure handler returns control back to the calling TSP operation. The set of possible failures is represented by the enumeration type *tsp_Failure*. The meaning of each failure value of type *tsp_Failure* is given below.

tsp_Failure_ERROR is a generic error code that can be raised by any TSP operation, e.g. if illegal argument values have been supplied to an operation.

tsp_Failure_DEADLOCK is raised if a deadlock is detected. Before this failure is signaled to the TSP client, TSP automatically performs a rollback. The resulting state of the store is *open* (see section 4.1).

tsp_Failure_STORE_FULL is raised if the maximum size of a store has been reached. It depends on the implementation of TSP whether the maximum size is limited by the number of objects or the total byte size of the store (or both).

tsp_Failure_OUT_OF_MEMORY is raised if TSP cannot allocate sufficient main memory to execute a TSP operation.

tsp_Failure_COMMIT is raised if the *tsp_commit* operation fails. TSP automatically triggers a rollback. This failure is only raised by backends that do not support a two-phase commit protocol.

4.3 Operations on Store Objects

TSP provides two different operations to create array objects and five operations to update and retrieve data from array objects. For each of these operations there exists an equivalent operation on byte array objects. The signatures of these operations look as follows:

```
tsp_OID
  tsp_newArray(tsp_Store s, tsp_Format f, tsp_Word nSlots);
tsp_OID
  tsp_newArrayInit(tsp_Store s, tsp_Format f,
                   tsp_Word nSlots, tsp_Word wInit);
tsp_Word
  tsp_getWord(tsp_Store s, tsp_OID array, tsp_Word iIndex);
void
  tsp_setWord(tsp_Store s, tsp_OID array, tsp_Word iIndex,
              tsp_Word w);
void
  tsp_getWords(tsp_Store s, tsp_Word nWords, tsp_OID array,
               tsp_Word iStart, tsp_Word *pwBuffer);
void
  tsp_setWords(tsp_Store s, tsp_Word nWords, tsp_OID array,
               tsp_Word iStart, tsp_Word *pwBuffer);
void
  tsp_moveWords(tsp_Store s, tsp_Word nWords,
                tsp_OID oidFrom, tsp_Word iFromStart,
                tsp_OID oidTo, tsp_Word iToStart);
```

Note that TSP provides built-in block move operations between persistent objects as well as between persistent objects and main memory data structures.

TSP also provides a set of operations that are applicable on all objects (arrays and byte arrays). The operation *tsp_nSlots* returns the number of slots of an object. The returned value does not reflect the allocation size because the slot size of array objects and byte array objects are different.

For the efficient implementation of some higher-level dynamic data structures, TSP exports the operation

```
tsp_OID tsp_resize(tsp_Store s, tsp_OID oid, tsp_Word nSlots);
```

which changes the number of slots of an object. Some store implementations support object resizing without restrictions. In this case, the returned object identifier is equal to the object identifier *oid* passed as an input parameter. If a store does not support growing or shrinking of objects, a new object with a fresh identifier is returned. In this case, *tsp_resize* is an efficient execution of the operation sequence to create a new object and to copy the slot values and the header information from the old object into the new object.

The format of an object as well as its mutability state can be retrieved and and changed by the following operations:

tsp_Format tsp_getFormat(tsp_Store s, tsp_OID oid);
***void** tsp_setFormat(tsp_Store s, tsp_OID oid, tsp_Format f);*
tsp_Bool tsp_isImmutable(tsp_Store s, tsp_OID oid);
***void** tsp_setImmutable(tsp_Store s, tsp_OID oid);*

While it is possible to dynamically change the format of an object, the mutability state of an object can be changed only once from mutable to immutable.

4.4 Main Memory Mapping of Store Objects

For some applications, like the implementation of string operations, the procedure call and addressing overhead incurred by repeated calls to object level operations as described in the previous section cannot be tolerated.

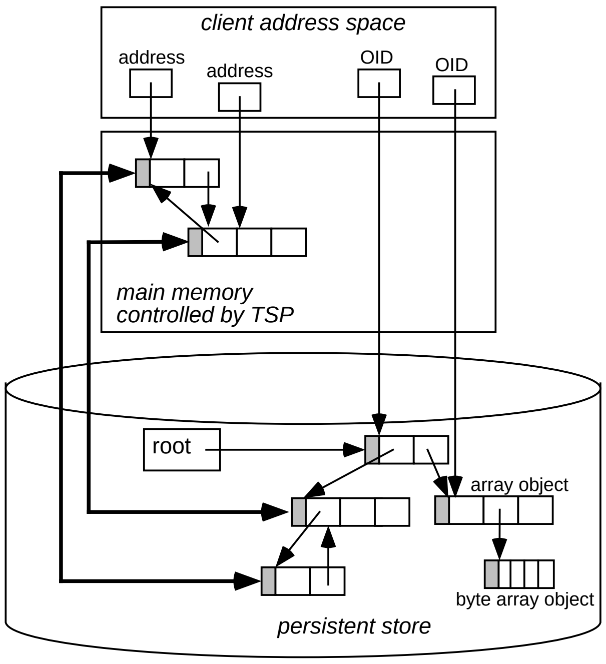

Figure 5: Storage Hierarchy and Object Identification

Therefore, TSP provides three additional operations (*tsp_openReadLock* and *tsp_openReadWriteLock* and *tsp_openRead*) to support fast object access. Each of these operations returns the address of a main memory representation of an object (see figure 5). The client can use the return value as a pointer to an array of *tsp_Tagged* or *tsp_Word* values if the object is an array, or as a pointer to an array of *tsp_Byte* values if the object is a byte array. Any open object must be closed explicitly using the operation *tsp_close* before calling *tsp_prepareCommit*. This deliberate restriction on the use of open objects greatly simplifies the implementation of TSP store adaptors while preserving much of the usefulness of main memory mapping.

If a persistent storage server support direct object access via main memory addresses, no main memory copy of an object has to be created to implement the *tsp_open* operations. Otherwise, a main memory copy is created by an intermediate layer of the TSP architecture. If *tsp_open* routines are called several times for the same object, TSP returns the same address on each call. It is not allowed to update an open object via TSP operations using its object identifier. Again, this design decision was made deliberately, balancing the complexity of TSP implementations against the additional burden put on TSP clients.

4.5 Management of Object Identifiers

A TSP client is allowed to store object identifiers in state variables outside the persistent store. As mentioned already in section 3.2, object identifiers are mutable. More specifically, they can be changed by a TSP implementation during the execution of *tsp_gc* (explicit garbage collection), *tsp_prepareCommit*, *tsp_commit* and *tsp_rollback*. Furthermore, an object creation operation (*tsp_new*...) can also trigger an implicit garbage collection.

TSP provides a mechanism to automatically remap modified object identifiers held outside the persistent store. To make use of this mechanism, a client has to implement a function that enumerates all relevant object identifiers. This client-defined function has to be installed explicitly each time a store is opened (*tsp_setEnumerator*). The type of an enumerator function and its argument function is given by the following type definitions:

```
typedef tsp_Tagged (*tsp_VisitFunction)
                  (tsp_Store s, tsp_Tagged tagged);
typedef void (*tsp_EnumFunction)
                  (tsp_Store s, tsp_VisitFunction visit);
```

The argument function *visit* is provided by a TSP backend. It maps old object identifiers to new object identifiers. A simple example of a client-defined enumerator function for an OID buffer is given below:

```
tsp_Tagged oidBuffer[10];
void enumerate(tsp_Store s, tsp_VisitFunction visit)
{  int i = 0;
   for(; i<10; i++) oidBuffer[i] = visit(s, oidBuffer[i]);}
```

The client-defined enumerator must not call TSP functions recursively.

A similar communication mechanism from the TSP backend to its client exists to notify a client prior to an implicit or explicit garbage collection:

```
typedef void (*tsp_GcHandler)(tsp_Store s);
void tsp_setGcHandler(tsp_Store s, tsp_GcHandler handler);
tsp_GcHandler tsp_getGcHandler(tsp_Store s);
```

The garbage collection handler of a newly created or opened store is initialized with the function *tsp_garbageCollect*. It can be overridden by the TSP client, for example, to close all open objects to improve the efficiency of the garbage collector or to display messages in an interactive application. Clients have to be aware that nested object creation operations may fail with an *tsp_Failure_STORE_FULL* exception.

4.6 Linearization of Object Store Subgraphs

As mentioned already in section 2, the external data representation TXR enables data exchange between arbitrary TSP stores. The operation *tsp_extern* transforms an object graph into a linear byte stream. Such a byte stream can be parsed with the operation *tsp_intern* to recreate an isomorphic object graph (object cycles and shared objects are preserved). The signatures of these TSP functions are defined as follows:

tsp_Word tsp_extern(tsp_Store s, tsp_Tagged tagged,
 ***void** *handle, tsp_WriteFunction wr);*
*tsp_Tagged tsp_intern(tsp_Store s, **void** *handle,*
 tsp_ReadFunction rd);

The stream functionality is factored out from the TSP by passing functions of type *tsp_WriteFunction* and *tsp_ReadFunction*, respectively, as function arguments to the *intern* and *extern* algorithms.

***typedef** tsp_Word (*tsp_WriteFunction)(**void** *handle,*
 *tsp_Byte *pbBuffer, tsp_Word nBytes);*

A *tsp_WriteFunction* is similar to the C low level *write* function defined on files. It takes a pointer to a buffer of bytes (*pbBuffer*) and writes *nBytes* starting at position **pbBuffer* to the stream identified by *handle*. A write function has to return the number of bytes that have been successfully written.

***typedef** tsp_Word (*tsp_ReadFunction)(**void** *handle,*
 *tsp_Byte *pbBuffer, tsp_Word nBytes);*

The semantics of a *tsp_ReadFunction* is similar to the Unix *read* function defined on files. It takes a pointer to a buffer of bytes (*pbBuffer*) and reads *nBytes* from the stream identified by *handle* into the buffer. A read function has to return the number of bytes successfully read.

In order to introduce client-defined linearization algorithms for client-defined atomic data values (identified by user-defined tags) or client-defined subtypes of array and byte array store objects (identified by user-defined format values), clients can use the following TSP types and TSP functions:

***typedef** tsp_Word (*tsp_ExternHandler)*
 (tsp_Store s, tsp_XDR xdr, tsp_OID oid,
 ***void** *handle, tsp_WriteFunction wr);*
***void** tsp_setExternHandler(tsp_Store s,*
 tsp_ExternHandler externHandler);
tsp_ExternHandler tsp_getExternHandler(tsp_Store s);

***typedef** tsp_OID (*tsp_InternHandler)*
 (tsp_Store s, tsp_XDR xdr, tsp_Tagged taggedMeta,
 tsp_Format f, tsp_Bool fImmutable, tsp_Word nSlots,
 ***void** *handle, tsp_ReadFunction rd);*
***void** tsp_setInternHandler(tsp_Store s,*
 tsp_InternHandler internHandler);
tsp_InternHandler tsp_getInternHandler(tsp_Store s);

The extern handler is called by a TSP implementation for each store object to be linearized while the intern handler is called to create a new object based on its header information and the contents of a stream.

Finally, some TSP extensions require a client-defined processing to be performed on each newly created object during *intern* operations, for example, to perform authentication checks in a distributed system. These checks can be installed as an *internVisit* operation for an individual store:

***typedef void** (*tsp_InternVisit)(tsp_Store s, tsp_OID oid);*
***void** tsp_setInternVisit(tsp_Store s, tsp_InternVisit visit);*
tsp_InternVisit tsp_getInternVisit(tsp_Store s);

5 Experience with TSP Implementations

In this section we report on our experience in implementing TSP backends by adaptors to existing object stores and extending the TSP functionality systematically by TSP -compliant layers.

In addition to the store adaptors described in the following subsections, our group at Hamburg University implemented two stand-alone object stores, called *Tymem* and *Tysin* that realize TSP with different performance characteristics. Both implementations are written in C and can be compiled on Unix, Linux, Windows, OS/2 and Macintosh OS platforms.

Tymem is a fast single user, main-memory based implementation of TSP biased towards applications that perform complex computations on rather small (up to 32MB) stores. The operation *tsp_commit* dumps the linear address space to a file and *tsp_openStore* reads the whole persistent store into main memory. Therefore, the size of a *Tymem* store is limited by the available virtual memory.

Tysin is a single user, disk-based store that performs data exchange between main and secondary memory on a segment basis. That is, on an object fault only the store segment(s) the persistent object resides in are brought into main memory and a *commit* only writes newly created or modified segments to secondary memory. Another difference between the two stores is the fact that *Tymem* garbage collects the whole object store on each invocation while *Tysin* limits most garbage collections to uncommitted store segments. Generally speaking, *Tysin* outperforms *Tymem* in all persistent applications that only work with a limited subset of the full persistent store. Furthermore, *Tysin* is also suitable for machines with poor virtual memory management support (like PC or Macintosh computers).

5.1 TSP Store Adaptors

The purpose of a store adaptor is to implement the functionality defined by TSP using the operations and data modeling features offered by a specific storage system. Our group has developed object store adaptors for the object oriented database system ObjectStore from Object Design [LLOW91] and for the proprietary object store of the University of St. Andrews [Bro89, BM91, BMM+92, Mun93] also used by the Napier88 [DCBM89] system. The University of Pisa is implementing a store adaptor for the O_2 [BDK92, CDKK85] object-oriented database engine. The following two subsections give a brief implementation overview of the existing TSP adaptors.

5.1.1 The ObjectStore Adaptor

ObjectStore poses very few restrictions on the kind of C++ or C values that can be stored persistently, for example, instances of classes, integers, strings, arrays and union typed values can be made persistent. ObjectStore supports object identification [OS993] through reference classes as well as through virtual memory addresses. The current ObjectStore adaptor uses the virtual address object identification scheme, i.e. values of type *tsp_OID* are virtual memory addresses valid in a single ObjectStore session. We now show how these ObjectStore features are used to implement the two basic TSP storage structures of polymorphic array and byte array objects.

A TSP (byte) array object is mapped to a C array with elements of type *tsp_Word* (*tsp_Byte*) preceded by a fixed-sized header.

The TSP type *tsp_Tagged* is mapped to an ObjectStore union type *Poly* defined as follows:

***union** Poly { tsp_Immediate immediate; Poly *obj;}*

Such a polymorphic value is either an immediate value or the address of another ObjectStore object. An array object with format *tsp_Format_TAGGED* is mapped to an array of *Poly* values. At runtime, the memory mapping mechanism of ObjectStore must be able to differentiate between virtual memory addresses and immediate values to change virtual addresses within objects in case the mapping of persistent objects into main memory has to be changed. Therefore, a user-defined discriminant function has to be associated with the user-defined union type *Poly* that inspects the actual value of the union typed value at runtime and tells ObjectStore which type it is, *tsp_Immediate* or *Poly**. The TSP discriminant function is give below.

tsp_Word Poly::discriminant( ){
  ***if** (IS_IMMEDIATE(imm)) **return** 1; **else return** 2; }*

The value returned by the function is an integer indicating the union's active field (1 for the first field and 2 for the second field). The implementations of the object operations of TSP are obvious given the above information. ObjectStore has no builtin garbage collector. Therefore, the TSP adaptor contains a copying garbage collector that is based on the possibility to structure ObjectStore databases into partitions.

5.1.2 The Napier Store Adaptor

The stable heap concept of the Napier Object Store is quite similar to the untyped TSP store model. A Napier store object is divided into three parts; an object header, an identifier and a data part. All object references must be stored in the identifier part. In addition, the client can store tagged immediates within the identifier part. The data part can be used to store data of any type. Thus, a TSP object, its header and the data part, are mapped to the identifier and data part of a Napier Object Store object. TSP does not separate object identifiers and immediate values. Therefore, all TSP client values must be stored in the tagged format. The minor difference of the two object concepts eases the implementation of the TSP object operations. For example, the read and write operations of TSP directly call the read and write operations of the Napier Object Store. Only the offsets of the addressed data items must be adapted.

The Napier Object Store uses a mechanism to handle cached object identifiers that is different from the TSP mechanism described in section 4.4. The client of a Napier Object Store has to install a *save* functions that is called before garbage collecting or stabilizing the store and a *restore* function that is called after these store activities. The purpose of these functions is to write identifiers to the store before store activities modify these object identifiers and to read the possibly changed object identifiers back into the cache after garbage collection and stabilization. The store adaptor maps the enumeration mechanism of TSP to the *save/restore* mechanism. This is done by implementing two *visit* functions passed as an argument to the TSP client enumerator function. This enumerator function is called prior to a stabilization/garbage collection with the visit function *visitSave*, that writes all cached object identifiers to a precreated object in the Napier Object Store. After a stabilization/garbage collection the enumerator function is called with a visit function *visitRestore* that reads the possibly modified object identifiers back into the cache. Furthermore, the store adaptor implements a mechanism to correctly handle open objects (see section 4.5) before and after garbage collection.

5.2 Extending the TSP Functionality

TSP has been designed to provide a standardized access to the core functionality of persistent stores. Systems where an extended functionality is required can enhance the TSP by additional operations. Some of these enhancements can be implemented in a way that does not depend on specific properties of the backend. In this case, the resulting software layers can be reused for multiple TSP stores. At Hamburg University, we have experimented with three such TSP extensions.

The first extension adds *persistent savepoints* to TSP by re-implementing the basic TSP operations and adding operations to rollback to a named savepoint. This feature makes it possible to structure long-running transactions in a way that simplifies application-controlled recovery. By rolling back to a named savepoint, the effects of a transaction on a persistent store are undone only partially. The store is responsible for keeping the necessary recovery logs and to re-establish the object store state of a specific savepoint.

The second extension adds *access control* on TSP objects. The security layer checks for each TSP operation whether a specific user has the right to perform this operation on a given persistent object. The underlying generic authorization scheme can be configured by clients of the security layer. The implementation utilizes callbacks and (optional) persistent access control structures attached to individual TSP objects.

A third extension provides *NFS access* to object stores by implementing a standard network file system (NFS [Sun90]) server using TSP operations. This server can be "mounted" directly by most commercial operating systems (Unix, MS-Windows, Macintos OS etc.). The NFS/TSP gateway maps operating systems files and directories directly and efficiently onto TSP store objects. This way, any tool that operates on operating system files, for example a text editor, an audio recorder or an image browser, can be used to access "exported " object store objects. Files managed by this NFS/TSP gateway can be manipulated through NFS file operations and TSP operations simultaneously. In particular for large multi-media objects, file access to recoverable persistent store objects is of high practical relevance.

6 Concluding Remarks

The main contribution of this paper is the in-depth description of an object store protocol that we regard as an excellent starting point for a *standardized* interface between the frontend and the backend of fully integrated persistent environments. In developing the TSP specification, its single-user implementations and its adaptors to commercial object stores available to other research groups, we hope to establish TSP as a widely accepted protocol within persistent language implementations. This would not only simplify the development of new persistent languages but also facilitate system measurements, comparative performance studies and interoperability experiments between multiple persistent languages.

References

[AB87] M.P. Atkinson and P. Bunemann. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2), June 1987.

[ABD+94] A. Albano, C. Brasini, M. Diotallevi, G. Ghelli, R. Orsini, and R. Rossi. A Guided Tour of the Fibonacci System. FIDE Technical Report Series FIDE/94/103, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, July 1994.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers, 1992.

[BM91] A.L. Brown and R. Morrison. A Generic Persistent Object Store. PPRR 2-91, Universities of Glasgow and St Andrews, 1991.

[BMM+92] A.L. Brown, G. Manietto, F. Matthes, R. Müller, and D.J. McNally. An Open System Architecture for a Persistent Object Store. In *Proceedings 25th Annual Hawaii International Conference on System Sciences*, volume 2, pages 766–776, January 1992.

[Bro89] A.L. Brown. Persistent Object Stores. PPRR 71-89, Universities of Glasgow and St Andrews, March 1989.

[Cat94a] R.G.G. Catell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1994.

[Cat94b] R.G.G. Cattell, editor. *Object Data Management*. Addison-Wesley Publishing Company, second edition, 1994.

[CDKK85] H.T. Chou, D.J. DeWitt, R.H. Katz, and A.C. Klug. Design and Implementation of the Wisconsin Storage System. *Software-Practice and Experience*, 15(10), October 1985.

[CMM+94] R.C.H. Connor, R. Morrison, D.S. Munro, D. Stemple, and S. Scheuerl. Concurrent Shadow Paging in the Flask Architecture. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Tarascon, France*, pages 16–37. Springer-Verlag, September 1994.

[DCBM89] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88 – A Database Programming Language? In *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, June 1989.

[GM95] A. Gawecki and F. Matthes. TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification. (Submitted for publication.), May 1995.

[ISO92] ISO. *Standard ISO 10303, Part1: Industrial Automation Systems – Product Data Representation and Exchange, Overview and Fundamental Principles*, 1992.

[KKD89] W. Kim, K. Kim, and A. Dale. Storage Management for Objects in EXODUS. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, Frontier Series. ACM Press, 1989.

[LLOW91] C. Lamb, G Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database. *Communications of the ACM*, 34(10):50–63, October 1991.

[MMS92] F. Matthes, R. Müller, and J.W. Schmidt. Object Stores as Servers in Persistent Programming Environments – The P-Quest Experience. FIDE Technical Report Series FIDE/92/48, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, July 1992.

[MMS95a] B. Mathiske, F. Matthes, and J.W. Schmidt. On Migrating Threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, June 1995.

[MMS95b] B. Mathiske, F. Matthes, and J.W. Schmidt. Scaling Database Languages to Higher-Order Distributed Programming. In *Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy*. Springer-Verlag, September 1995.

[MS88] J.E.B. Moss and S. Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable Shared Object Interface. In K.R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, number 334 in Lecture Notes in Computer Science. Springer-Verlag, September 1988.

[MS92] F. Matthes and J.W. Schmidt. Definition of the Tycoon Language TL – A Preliminary Report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[MS93] F. Matthes and J.W. Schmidt. System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways. In P.P. Spies, editor, *Proceedings of Euro-Arch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.

[MS95] F. Matthes and J.W. Schmidt. Persistent Threads. In M.P. Atkinson, editor, *FIDE: Fully Integrated Data Environments*. Springer-Verlag, 1995.

[Mül91] R. Müller. Language Processors and Object Stores: Interface Design and Implementation. Master's thesis, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, November 1991. (In German).

[Mun93] D.S. Munro. *On the Integration of Concurrency, Distribution and Persistence*. PhD thesis, Department of Mathematical and Computational Sciences, University of St. Andrews, Scotland, 1993.

[OS993] Object Design Inc., Burlington, MA. *ObjectStore Release 3 for Unix Systems: Reference Manual*, 1993.

[RHB+90] John Rosenberg, Frans Henskens, Fred Brown, Ron Morrison, and David Munro. Stability in a Persistent Store Based on a Large Virtual Memory. In J. Rosenberg and J.L. Keedy, editors, *Security and Persistence*, pages 229–245. Springer-Verlag, 1990.

[Sun90] Sun Microsystems. Network File System: Version 2 Protocol Specification. Technical report, Sun Microsystems, 1990.