Tempest Interface Specification

Revision 1.2.1
February 22, 1995

Steven K. Reinhardt

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706
stever@cs.wisc.edu

1 Introduction

This document describes *Tempest*, an architectural user/system interface for distributed-memory parallel systems. By providing user-level access to both messaging and memory-management functions, Tempest not only supports message passing and shared memory, the two dominant parallel programming models, but allows users to construct hybrid models as well. The role of Tempest in parallel programming is similar to that of an instruction set architecture in uniprocessor programming:

- Tempest provides a relatively low-level interface between user-level software and the system. The primary users of Tempest are developers of programming tools, i.e., compilers, libraries, and run-time systems. Applications are typically developed using the high-level languages and constructs supported by these tools, resorting to direct use of Tempest only for hard-core performance tuning.
- Tempest is designed to support a range of implementations. Applications and tools using Tempest are portable across multiple platforms, from networks of workstations or personal computers to high-performance parallel processors. These platforms use a variety of software and hardware techniques to support Tempest at several different cost/performance points.
- Tempest represents a contract between system designers and software developers. This contract allows the system designers to focus on efficient implementation of the set of operations in the interface, while the software developers build on these operations to create powerful and flexible programming tools and applications.

The most unusual feature of Tempest is *fine-grain memory access control*. An access tag is associated with every block of memory at a granularity typical of a hardware cache block (e.g., 32-128 bytes). Loads and stores that conflict with the access tag invoke a user-specified handler that can perform arbitrary operations to resolve the conflict. The ability to dynamically manage data access at a fine grain allows data migration and replication without relying on compile-time analysis or a restricted programming model to guarantee correctness. This migration and replication can be performed without renaming using Tempest's *virtual memory management* operations. To effect data transfer between nodes, Tempest provides two types of message-passing. *Fine-grain messaging* provides the short, low-latency messages required to implement cache coherence protocols and support fine-grain parallelism. *Bulk data transfer* operations are optimized to provide high bandwidth for large messages. *Timers* and *thread management* complete the list of Tempest features.

Tempest was originally proposed (at a very high level) in Reinhardt, et al. [4], which gives further motivation for the interface, a description of how it can be used to implement "vanilla" cache-coherent shared memory, and an example of an application-specific optimization. Readers interested in these topics are referred to that paper.

This document is a formal specification of the Tempest interface. The specification is divided into two parts: Section 2 describes the general execution model of a Tempest application and Section 3 documents the specific interface functions for the C programming language. In addition to the interface specification, comments labeled “**Rationale**” and “**Implementation note**” are provided to give insight into the motivation for certain features and to suggest implementation approaches, respectively.

2 Execution model

The Tempest architecture assumes a host that consists of many processing nodes, where each node contains one or more processors connected to a single memory module (that is, sharing a single physical address space). Communication between nodes occurs only through message-passing and the (as yet undefined) global operations (e.g., barriers). A Tempest application has a distinct virtual address space on each processing node. The SPMD (“single program, multiple data”) model is used, i.e., the same program text is loaded at the same location in every address space, though each processor executes that text independently. Each address space may also contain other per-node private segments for data and stacks. In addition, a contiguous segment at the same location in each address space is designated the *user-managed virtual segment*. Within this segment, the user maps virtual pages to physical memory, handles accesses to unmapped pages, and controls the accessibility of mapped memory at a fine granularity. Providing user-level control over the same virtual address region on every node is the basis of constructing a transparent single-address-space execution environment with user-defined semantics.

Tempest provides fine-grain memory access control by associating a *block access tag* with every aligned 2^k-byte region of memory (a *memory block*). The value of k is implementation-dependent, but is typically five, six, or seven; that is, memory blocks are typically 32, 64, or 128 bytes. The Tempest interface is designed so that code written assuming some block size b will be portable across all implementations whose block size is smaller than or equal to b. The actual block size supported by an implementation is referred to as the implementation’s *minimum block size*, and blocks of the minimum block size are called *minimal blocks*.

Access	**Tag**			
	Invalid	**Busy**	**ReadOnly**	**Writable**
load	fault	fault	return data	return data
store	fault	fault	fault	write data

Table 1: The result of load and store operations according to the referenced block’s access tag.

Each block access tag has one of four values: Invalid, Busy, ReadOnly, and Writable. A block whose tag is Invalid or Busy is referred to as *invalid*, while blocks tagged ReadOnly or Writable are called *valid*. (Invalid and Busy have the same access semantics, but can be used by software to encode protocol information; e.g., whether or not a request is pending.) Loads and stores are checked against the value of the referenced block’s access tag; conflicting accesses cause a *block access fault* (see Table 1). The fault suspends the accessing thread and invokes a user-level handler to process the fault. The handler typically performs some protocol actions to make the access permissible. Once access is allowed, the block’s tag is changed and the faulting thread is resumed.

Each address space (and thus each node) can support multiple threads of execution. These threads may execute concurrently on multiprocessor nodes. One of the threads is distinguished as the *protocol thread*. All other threads are referred to as *computation threads*. The management of computation threads (creation, scheduling, etc.) is outside the domain of Tempest. The protocol thread is scheduled by Tempest and exists solely to execute user-defined *handler functions* to process *protocol events*: network message arrivals, timer expirations, and the page faults and block access faults of computation threads. Handler functions are executed sequentially, i.e., if a protocol event occurs while the protocol thread is executing a handler function, the event will be queued until the current handler function completes. If multiple events are queued, block access faults and page faults are given first priority, followed by timer expirations, and then message arrivals.

> **Rationale:** By dedicating a single thread for protocol processing and running handler functions to completion, the need for locking on protocol data structures is reduced or eliminated. If concurrency is desired in processing protocol events, the protocol thread's handler function can hand off tasks to computation threads.
>
> **Rationale:** Event prioritization is based on the following observations:
>
> - Message arrivals are beyond the control of the local node, so any event with lower priority may be subject to starvation. This is avoided by giving message arrivals lowest priority.
> - Servicing of faulting accesses should be given high priority to maximize application throughput. The number of concurrent page and block access faults is limited by the number of concurrently-executing computation threads, so starvation of other event types is not a problem.

Only the loads and stores of computation threads are guaranteed to be checked against the access tags. Normally, the protocol thread only references tagged memory indirectly via Tempest functions; the behavior of a direct protocol thread access that conflicts with the block's tag is undefined.

The type of protocol event determines the user-level handler function that is executed by the protocol thread. The handler for a message arrival is chosen by the sender and encoded in the message header (as in Active Messages [6]). The handler for a timer expiration is specified when the timer is initialized. Handlers for page faults and block access faults are registered locally by the application. All page faults are serviced by a single handler. The handler invoked for a block access fault is determined by the combination of the access type (load vs. store), the tag value, and the memory page on which the referenced block is located. Specifically, the user associates a small integer (the *page mode*) with every page, and registers a set of five handlers (one for each of the "fault" cases in Table 1) for each page mode.

> **Rationale:** The ability to associate different sets of handlers with different memory blocks facilitates the use of multiple protocols within a single application. Associating handlers with pages rather than individual blocks trades a small loss of flexibility for a large decrease in storage overhead. (Actions can still be specialized at block granularity in software.) Use of the page mode value rather than allowing a separate set of handlers for each page further reduces storage overheads.

3 The Tempest interface for C

This section defines a standard C user interface for Tempest to provide source-level application compatibility across all Tempest implementations. Implementations will typically provide a user library that bridges the gap between Tempest and the native operating system. A Tempest implementation may involve more than a library with which the application is linked. For example, the application source may be preprocessed to convert Tempest function calls to some intermediate form, or the compiled application may be postprocessed to insert code that provides fine-grain access control [5]. The key characteristic is that an automated process is provided that converts interface-compliant source into a functioning program.

The following subsections specify the operations provided by the interface, grouped by function: virtual memory management, fine-grain memory access control, fine-grain messaging, bulk data transfer, timers, and thread management.

3.1 Virtual memory management

As described in Section 2, Tempest gives user-level control over a region of the virtual address space on each node. This region, known as the *user-managed virtual segment*, is located at the same address in every address space and is at least 1 Gbyte in size. Physical memory allocation and address translation are performed on the basis of pages. Pages mapped in the user-managed virtual segment are referred to as *user-managed* pages. Only user-managed pages support block access tags. Each user-managed page must be assigned a *page mode number*, which determines the set of block access fault handlers that are invoked for block access faults on that page.

3.1.1 Page size

```
#define TPPI_PAGE_SHIFT             implementation-specific
#define TPPI_PAGE_SIZE              (1 << TPPI_PAGE_SHIFT)
```

The page size in bytes is exported in the constant TPPI_PAGE_SIZE. TPPI_PAGE_SHIFT is the width in bits of a page offset (i.e. $\log_2$(TPPI_PAGE_SIZE)).

lmplementation note: The Tempest page size should be as small as possible to avoid memory fragmentation. Typically it is the same as the platform's MMU page size; e.g., the SPARC MMU page size is 4K so all existing implementations (which are all SPARC-based) have a 4K page size.

3.1.2 Page modes

```
typedef implementation-specific TPPI_PageMode;
#define TPPI_NUM_PAGE_MODES         implementation-specific
#define TPPI_MAX_PAGE_MODE          (TPPI_NUM_PAGE_MODES - 1)
```

Page mode numbers are small consecutive integers starting at 0. TPPI_PageMode is an unsigned integer type of implementation-defined size used to hold page mode numbers. The constant TPPI_NUM_PAGE_MODES indicates the number of page modes supported by the implementation, and TPPI_MAX_PAGE_MODE indicates the largest page mode number.

3.1.3 Page allocation and deallocation

```
int TPPI_alloc_and_map(void *pg, TPPI_PageMode mode, TPPI_BlkAccTag acc, TPPI_NodeId home,
                        void *usr_ptr);
```

Allocates a page of physical memory and maps it in the user-managed virtual segment at the page-aligned address pg. By definition, the allocated page is a user-managed page. The page mode (for block access fault handler selection) is set according to mode. The block access tags for all blocks on the page are initialized to acc (see Section 3.2.1). The home and usr_ptr fields are not interpreted by the system, but are intended to hold the node identifier (see Section 3.3.1) of the page's directory node and a pointer to a per-page protocol data structure, respectively. The home and usr_ptr values can be retrieved via function calls (using the virtual address as a key) and are automatically provided to block access fault handlers for the page.

The return value is 1 if successful, 0 if unsuccessful. The call will fail if there is insufficient physical memory, pg is not aligned to the page size, a mapping already exists for the virtual address pg, or pg is not within the user-managed virtual segment.

> **lmplementation note:** Page-level protection may be used in the place of a true fine-grain access control mechanism if, whenever the user invokes a tag-modifying block operation, the specified block length is always equal to or greater than the page size. Because pages are always initialized with the same tag on every block, this optimization can be performed optimistically; that is, the fine-grain access control mechanism need not be used for a given page until the first time the user invokes a tag-modifying block operation with a block length smaller than the page size.

int TPPI_unmap_and_free(void *pg);

Removes the mapping for user-managed page pointed to by the aligned address pg.

int TPPI_remap(void *old_pg, void *new_pg);

Removes the mapping for user-managed page old_pg and remaps the physical page to the address new_pg, which must also be in the user-managed virtual segment. The block access tags, page mode, home node ID, and user pointer are unchanged. Both old_pg and new_pg must be page-aligned.

> **Rationale:** While TPPI_unmap_and_free followed by TPPI_alloc_and_map has a similar effect, TPPI_remap differs in two significant ways. First, the same physical page is kept, so the data is not lost. Second, TPPI_remap keeps the page continuously under the ownership of the application; the TPPI_unmap_and_free/TPPI_alloc_and_map sequence is non-atomic so it is possible that another application could perform an allocation in the middle causing the TPPI_alloc_and_map to fail due to insufficient memory.

3.1.4 Page fault handlers

typedef void (*TPPI_PageFaultHandlerPtr)(void *va, int pc, int is_write);

void TPPI_register_page_fault_handler(TPPI_PageFaultHandlerPtr fn);

Registers fn as the user's page fault handler. When a computation thread accesses an unmapped virtual address in the user-managed virtual segment, the Tempest implementation will notify the user by causing the protocol thread to invoke this function. The handler will be invoked as

void (*fn)(void *va, int pc, int is_write)

where va is the unmapped address that was accessed and pc is the program counter of the load or store that caused the fault. The is_write parameter is non-zero if the access was a store, or zero if it was a load. The faulting thread is suspended until TPPI_resume_va is called (see Section 3.6), either by the page fault handler, a future message handler, or a different thread.

Before resuming the faulting access, the page fault handler may directly request needed data from a remote node or it may simply initialize all blocks on the page to TPPI_Blk_Invalid. In the latter case, when the access is retried after the thread is resumed it will generate a block access fault. This approach may be favored because it keeps protocol-specific code out of the page fault handler.

> **Rationale:** The page fault handler will typically use TPPI_alloc_and_map to add a page at the desired address. It may need to send a request to a remote node to determine the appropriate page mode; in this case, the TPPI_alloc_and_map and the resumption of the faulting thread will be performed by the response message handler.

3.1.5 Retrieving per-page information

```
typedef struct {
        implementation-specific
} TPPI_PageInfo;
```

int TPPI_get_page_info(void *va, TPPI_PageInfo *info_ptr);

Provides information about the virtual page containing the arbitrarily-aligned address va in the user-managed virtual segment. The return value is 0 if the page is not mapped, 1 if it is mapped, and -1 if there is an error (because va is not in the user-managed virtual segment or info_ptr is not suitably aligned). If info_ptr is non-null and the return value is 1, the mode, usr_ptr, and home values provided when the page was mapped are written to the fields of the same name in the structure pointed to by info_ptr. The exact definition of TPPI_PageInfo is implementation-dependent, but it must contain at least these three fields: TPPI_PageMode mode, void *usr_ptr, and TPPI_NodeId home.

```
void *TPPI_get_page_user_ptr(void *va);
```

Returns the usr_ptr pointer supplied to TPPI_alloc_and_map when the page containing the arbitrarily-aligned address va was mapped. The return value is undefined if the page is not in the user-managed virtual segment or is not mapped.

```
TPPI_PageMode TPPI_get_page_mode(void *va);
```

Returns the mode value supplied to TPPI_alloc_and_map when the page containing the arbitrarily-aligned address va was mapped. The return value is undefined if the page is not in the user-managed virtual segment or is not mapped.

```
TPPI_NodeId TPPI_get_page_home(void *va);
```

Returns the home value supplied to TPPI_alloc_and_map when the page containing the arbitrarily-aligned address va was mapped. The return value is undefined if the page is not in the user-managed virtual segment or is not mapped.

3.2 Block access control

The semantics of block access tags are discussed in Section 2. Tag values are initialized during page allocation (see Section 3.1.3). Note that tags can also be modified as a side-effect of sending or receiving blocks using the Ba item type (see Section 3.3).

3.2.1 Access tag values

```
typedef enum {
        TPPI_Blk_Busy, TPPI_Blk_Invalid, TPPI_Blk_ReadOnly, TPPI_Blk_Writable
} TPPI_BlkAccTag;

#define TPPI_NUM_BLK_ACC_TAGS    4

#define TPPI_MAX_BLK_ACC_TAG     (TPPI_NUM_BLK_ACC_TAGS - 1)
```

The TPPI_BlkAccTag type enumerates the possible access tag values for a block. The enumeration constants will be named as specified and valued from 0 to 3 inclusive, but their ordering is implementation-dependent. The constants TPPI_NUM_BLK_ACC_TAGS and TPPI_MAX_BLK_ACC_TAG indicate the number of supported access tags and the largest access tag value, respectively.

3.2.2 Tag block size

```
#define TPPI_TAG_BLK_SHIFT          implementation-specific

#define TPPI_TAG_BLK_SIZE           (1 << TPPI_TAG_BLK_SHIFT)
```

The implementation's minimum tag block size in bytes (see Section 2) is exported in the constant TPPI_TAG_BLK_SIZE. TPPI_TAG_BLK_SHIFT is the width in bits of an offset within a block (i.e., $\log_2$(TPPI_TAG_BLK_SIZE)).

3.2.3 Specifying memory blocks

A memory block is specified with two parameters: an address and a length in bytes. These appear as a pair of arguments: `void *blk_va, int blk_len`. Whenever an address/length pair is used to specify a block as the target of a Tempest operation, the length must be a power of two and must be greater than or equal to the implementation's minimum block size. The address does not have to be aligned; it may point anywhere within the block.

> **Rationale:** For systems with hardware support, it is trivial to ignore unused address bits, so forcing the user to align addresses introduces unnecessary overhead. For software-based systems, the potential exists for address alignment to be inlined at the call site, with common sub-expression elimination allowing a single alignment operation to serve for multiple Tempest function calls. Assuming these optimizations, there is little performance advantage in forcing the user to perform alignment.

If the specified length is greater than the minimum block size, the block is called a *superblock*. Operations on superblocks are subject to the following constraints:

- The operation should be viewed as a non-atomic series of operations on the constituent minimal blocks. The atomicity guarantees of Section 3.2.6 only apply to the minimal-block operations.
- If a tag change operation (of type TPPI_BlkTagChange) is applied to a superblock, it must be a valid tag change for each of the constituent minimal blocks. Note that this does not mean that all of the minimal blocks must have the same original tag (though this is likely to be the case). For example, TPPI_Blk_Invalidate could be applied to a superblock in which some minimal blocks are tagged TPPI_Blk_ReadOnly and others TPPI_Blk_Writable.

Otherwise, it should be transparent to the user whether an operation is applied to a single minimal block or a superblock.

3.2.4 Reading access tags

```
TPPI_BlkAccTag TPPI_get_blk_acc(void *va);
```

Returns the block access tag associated with the block containing va. The result is undefined if the address is not in the user-managed virtual segment or is not mapped.

3.2.5 Changing access tags

```
typedef enum {
        TPPI_Blk_Validate_RW, TPPI_Blk_Upgrade_RW, TPPI_Blk_Validate_RO,
        TPPI_Blk_Downgrade_RO, TPPI_Blk_Invalidate, Blk_Mark_Busy,
        TPPI_Blk_No_Tag_Change, TPPI_Blk_Invalid_To_Busy,
        TPPI_Blk_Busy_To_Invalid
} TPPI_BlkTagChange;
```

Block access tag modifications are made using the TPPI_BlkTagChange constants. These not only specify the desired tag value but also imply the current value of the tag, as specified in Table 2. If the user applies a tag change operation to a block whose tag is not one of those implied by the operation (i.e., the operation does not appear in the row corresponding to that tag in Table 2), the resulting state of the block is indeterminate. Note that TPPI_Blk_Invalidate, TPPI_Blk_Mark_Busy, and TPPI_Blk_Validate_RW (and of course TPPI_Blk_No_Tag_Change) can be applied to any block, regardless of its initial state, though in some cases the same tag change may be performed more efficiently using a different operation.

> **Rationale:** Reducing access to a block typically requires that the block be flushed from any hardware caches, while increasing or not changing block access does not. These cache flushes can be very expensive and need to be avoided when possible. A simple "set tag" function is not sufficient to identify

Current Tag	New Tag			
	Invalid	**Busy**	**ReadOnly**	**Writable**
Invalid	No_Tag_Change, Invalidate	Invalid_To_Busy, Mark_Busy	Validate_RO	Validate_RW
Busy	Busy_To_Invalid, Invalidate	No_Tag_Change, Mark_Busy	Validate_RO	Validate_RW
ReadOnly	Invalidate	Mark_Busy	No_Tag_Change	Upgrade_RW, Validate_RW
Writable	Invalidate	Mark_Busy	Downgrade_RO	No_Tag_Change, Validate_RW

Table 2: Block tag change enumeration values (type TPPI_BlkTagChange). All values are prefixed by TPPI_Blk_, e.g., TPPI_Blk_No_Tag_Change. Where two values are listed in a single entry, the first is preferred.

when flushes are necessary, and requiring the implementation to look up the current tag before it is changed (to determine is a flush is required) may also be expensive. The current tag state is usually implied by the user protocol state, so the user code typically has enough information to supply the tag change operation with no extra overhead.

void TPPI_change_blk_acc(void *blk_va, int blk_len, TPPI_BlkTagChange chg);

Changes the access tag of the block specified by (blk_va, blk_len).

void TPPI_change_blk_acc_and_copy(void *blk_va, int blk_len, TPPI_BlkTagChange chg, void *from);

Copies data from memory starting at from to the block specified by (blk_va, blk_len) and changes the tag of the block according to chg.

3.2.6 Atomicity of data access and tag changes

Threads may be executed concurrently on implementations with multiprocessor nodes, so while one thread is in the middle of a tag change, other threads may issue loads and stores. Tempest operations that combine data transfer and access tag changes (including TPPI_change_blk_acc_and_copy , send_*Ba*, and recv_Ba) provide the following useful semantics:

- If data is read from a block and the block's access is downgraded from Writable, the block data that is read is guaranteed to reflect all writes that complete before the tag change.
- If data is written to a block and the block's access is upgraded, any load or store that does not fault but would have faulted given the previous access tag is guaranteed to be performed after the block's contents are updated with the new data.

3.2.7 Block access fault handlers

```
typedef void (*TPPI_BlkAccFaultHandlerPtr)(void *va, void *user_ptr, TPPI_NodeId home);
```

```
void TPPI_register_blk_acc_fault_handler(TPPI_PageMode mode, TPPI_BlkAccTag tag, int acc,
                                TPPI_BlkAccFaultHandlerPtr fn);
```

Registers function fn as the block access fault handler for accesses of type acc (which should be one of the defined constants TPPI_ReadAccess or TPPI_WriteAccess) to blocks tagged with tag on pages of mode mode. The handler will be invoked as

```
void (*fn)(void *va, void *usr_ptr, TPPI_NodeId home)
```

where va is an address within the block on which the faulting access was performed and usr_ptr and home are the values supplied to TPPI_alloc_and_map when the page containing va was mapped. The actual address that was accessed by the faulting thread and va will be in the same minimal block, but are not necessarily related otherwise.

Rationale: Implementations using hardware external to a commodity processor will only observe the cache miss that results from a faulting access, not the faulting access itself. In this case, the relationship between the observed address and the accessed address will be determined by the processor implementation.

Only five of the eight tag/acc combinations are meaningful (see Table 1); specifying handlers for the other three (TPPI_Blk_ReadOnly/TPPI_ReadAccess, TPPI_Blk_Writable/TPPI_ReadAccess, and TPPI_Blk_Writable/TPPI_WriteAccess) may have undesirable implementation-dependent effects and should be avoided. (Ideally, the implementation will detect attempts to specify handlers for the other cases and warn the user.)

3.3 Fine-grain messaging

Fine-grain messaging provides low-overhead message sending and reception, optimized for short message lengths.

Rationale: Both cache coherence protocols and fine-grain parallel applications employ short asynchronous messages whose contents are immediately consumed on receipt (e.g., cache miss or remote read requests and responses). Much of the message data originates in the sender's registers and is consumed in the receiver's registers. The memory-to-memory transfers provided by most message-passing models (and by Tempest's bulk data transfer operations) are inappropriate for these applications since both the management of memory buffers and the need to copy data into and out of these buffers add significant overhead.

Tempest's fine-grain messaging facility is based on Active Messages [6]. In the Active Message model, the first word of every message is the starting program counter of the handler to be executed at the receiver. Messages are queued and the handlers are executed serially by the protocol thread.

3.3.1 Node identifiers

typedef *implementation-specific* TPPI_NodeId;

```
unsigned TPPI_num_nodes;
```

```
TPPI_NodeId TPPI_self_address;
```

TPPI_NodeId is an unsigned integer type of implementation-defined size used to hold node identifiers. Node identifiers are in the range 0 to *n*-1 for *n*-node systems. Two integer variables, TPPI_num_nodes and TPPI_self_address, provide the number of available nodes and the local node's identifier, respectively.

3.3.2 Sending

typedef void (*TPPI_MessageHandlerPtr)(TPPI_NodeId src, int size);

void TPPI_send_*typelist*(TPPI_NodeId dest, TPPI_MessageHandlerPtr pc, *arglist*);

This set of functions sends a message to the specified node, where the message will be handled by executing code starting at the specified program counter. The body of the message is constructed using the specified (possibly empty) item list. In the current C binding, the item list specification is split: the types of the items in are encoded in a string that is part of the function name, while the parameters describing the items are part of the argument list. A given item may require more than one parameter.

Rationale: Abstractly, TPPI_send is a polymorphic function that takes an arbitrary number of arguments selected from a set of types (word, memory block, and memory region). Unfortunately, C does not support this polymorphism. The C++ binding (when complete) will have a single TPPI_send function that is overloaded to support all possible message formats.

The following item types are available (with the type string given in parentheses):

- *Word (W).* A single machine word is sent. The corresponding parameter is the word value, of type int.
- *Block with access change (Ba).* The contents of a memory block are sent, and the memory block's access tag is modified. The corresponding parameters are the block specifier (void *blk_va, int blk_len) (see Section 3.2.3) and the tag change (type TPPI_BlkTagChange) (see Section 3.2.5).
- *Region (R).* The contents of a region of memory are sent. The region must start on a word boundary and contain an integral number of words. The corresponding parameters are the region start address (type void *) and the region length in bytes (type int). The region length must be a multiple of the word size.
- *Forward (F).* Data from the current received message is sent. This option is only valid when the send is called in the context of a message handler. The corresponding parameter is the number of bytes to forward (type int), which must be a multiple of the word size.

For example, the following call sends a word of data (word) along with a memory block of size blk_len at address blk_va, atomically changing the block's tag from ReadOnly to Invalid:

```
TPPI_send_WBa(dest, handler_pc, word, blk_va, blk_len, TPPI_Blk_Invalidate);
```

As a syntactically special case, the '_' in the function name is elided when a message with no body is sent, e.g., TPPI_send(dest, pc).

The message body is constructed by concatenating data items, in the specified order, into an untyped stream of words.

3.3.3 Receiving

On the receiver, the system logically queues the message until the protocol thread is idle. The sender-specified function is invoked with two parameters: the source node (type TPPI_NodeId) and the size of the message body in bytes (type int). The message body is provided as a logical queue of words. Data is read from this queue and consumed using the following calls, which correspond to the types available for sending:

int TPPI_recv_W();

The next word is returned.

void TPPI_recv_Ba(void *blk_va, int blk_len, TPPI_BlkTagChange chg);

The next blk_len bytes are read from the queue and written to the memory block specified by (blk_va, blk_len) (see Section 3.2.3), whose access tag is changed (see Section 3.2.5).

void TPPI_recv_R(void *va, int len);

The next len bytes of data are read from the queue and written to the specified region of memory. The region must start on a word boundary and contain an integral number of words.

In addition, message data can be consumed using a “forwarded block” item in a send operation. Note that even though the send and receive operations use the same types, the message body is transferred as a typeless word stream, so the types used to send and receive a particular message do not need to match. However, the receive handler must consume the entire message body. If a receive handler leaves data in the message queue when it terminates, some implementations may interpret this data as part of a separate message. The resulting behavior is undefined.

3.3.4 Message size limit

#define TPPI_MAX_AM_BYTES *implementation-specific*

A Tempest implementation will typically have an upper bound on the size of message that can be supported in terms of the number of bytes in the message body. This upper bound is exported in the constant TPPI_MAX_AM_BYTES and is guaranteed to be at least (TPPI_TAG_BLK_SIZE + 16).

Rationale: This minimum size allows for a block and 16 bytes of control information (e.g., four 32-bit words, or two 32-bit words and a 64-bit address).

lmplementation note: Tempest provides an Active Message interface without enforcing an Active Message implementation. In an actual Active Message implementation, the entire message is put into a single packet. On a system that cannot support a (TPPI_TAG_BLK_SIZE + 16)-byte payload in a single packet, packetization and reassembly must be supported, but messages that are “small enough” may still be handled in a true Active Message fashion.

3.4 Bulk data transfer

Bulk data transfer provides high-bandwidth, connection-oriented, memory-to-memory data movement between nodes.

Rationale: A memory-to-memory transfer model is desirable because it simplifies system flow control and buffering issues by inherently providing buffer space on both the sender and receiver, and it can be efficiently supported with typical DMA hardware. A connection-oriented model allows connection set-up overhead to be amortized over multiple transfers when a repetitive communica
tion pattern exists.

lmplementation note: All memory-to-memory transfers could be implemented on top of a suitable Active Messages layer.

3.4.1 Channel allocation

typedef void (*TPPI_ChannelHandlerPtr)(NodeId, int channel);

int TPPI_set_channel_src(TPPI_NodeId dest, TPPI_ChannelHandlerPtr fn);

int TPPI_set_channel_dst(TPPI_NodeId src, int channel, void *buffer, int bytes,
                          TPPI_ChannelHandlerPtr fn);

Channel allocation requires allocation of an endpoint on both the source (sending) and destination (receiving) nodes. The source node must first call TPPI_set_channel_src() to obtain a channel ID number.[1] The arguments are the destination node and a pointer to a function which will be invoked at the completion of each send. The sender must communicate the returned channel ID to the destination node (typically via an active message). The destination node then calls TPPI_set_channel_dst() to initialize the receiving end, passing in the source node ID, the channel ID from the source, the address and length of a receive buffer, and a pointer to a function that will be invoked at the completion of each receive.

Blizzard Note: The buffer must be a multiple of four bytes in length and four-byte-aligned.

Both the send and receive callbacks are invoked with the ID of the corresponding node and the channel ID. In either case, a null function pointer may be provided in which case no callback will be performed. On the source node, invocation of the callback means only that the send buffer can be reused; it does not imply that the data has been received at the destination.

TPPI_set_channel_dst() and sends an active message back to the source to notify it that the endpoint has been established. The source node) to poll for the establishment of the destination endpoint. It will return non-zero only after the arrival of the notification message.

3.4.2 Sending data

void TPPI_channel_send(TPPI_NodeId dest, int channel, void *buffer, int bytes);

The source node calls TPPI_channel_send() to initiate a data transfer of bytes bytes starting at the pointer buffer. The transfer may be asynchronous; the send callback, if any, will be invoked when the buffer memory may be reused. The number of bytes specified in the send must exactly match the number specified by the destination node in its call to TPPI_set_channel_dst().

Blizzard Note: The buffer must be a multiple of four bytes in length and four-byte-aligned.

int TPPI_is_channel_ready(TPPI_NodeId src, int channel);

The destination node may call TPPI_is_channel_ready() to poll for the arrival of data (in lieu of specifying a receive callback function). This function will return non-zero when the destination has received the number of bytes specified in its call to TPPI_set_channel_dst(). It will continue to return non-zero until the endpoint is reset via TPPI_reset_channel().

void TPPI_reset_channel(TPPI_NodeId src, int channel);

The destination node must call TPPI_reset_channel() to reset the receive endpoint of the channel after each data transmission before the source can perform another send. The destination and source nodes must synchronize to guarantee that the destination has called TPPI_reset_channel() before the source calls TPPI_channel_send().

TPPI_reset_channel() and sends an active message back to the source to notify it that the endpoint has been reset. The source node) to poll for this event. It will return non-zero only after the arrival of the notification message.

3.4.3 Channel deallocation

void TPPI_destroy_channel_src(TPPI_NodeId dest, int channel);

void TPPI_destroy_channel_dst(TPPI_NodeId dest, int channel);

As with allocation, both the source and destination nodes must explicilty deallocate their endpoints. Results are unpredictable if either endpoint is deallocated before all of the data that sent on the channel has been received at the destination.

1. The channel ID may be relative to a source/destination pair, i.e., distinct channel IDs are only required when there a multiple active channels between a given source and destination.

3.4.4 User pointers

Each endpoint (source and destination) contains storage for an arbitrary pointer so that the user may associate application-specific structures with the channel. These functions provide access to that storage.

3.4.5 Transfer size limit

```
#define TPPI_MAX_CHANNEL_BYTES        implementation-specific
```

The constant TPPI_MAX_CHANNEL_BYTES indicates the maximum number of bytes that can be transferred through a channel between calls to TPPI_reset_channel().

3.5 Timers

Efficient timers are useful for implementing protocol time-outs and providing flexible forward-progress guarantees.

```
typedef void (*TPPI_TimerHandlerPtr)(void *user_ptr);
void TPPI_schedule_timer(int ticks, TPPI_TimerHandlerPtr fn, void *user_ptr);
```

Schedules a timer event for ticks units of time in the future. The units for ticks are implementation-dependent. After the timer event occurs, the handler function fn will be invoked by the protocol thread with the single argument user_ptr.

3.6 Thread management

```
void TPPI_resume_va(void *blk_va, int blk_len);
```

Resumes the set of threads suspended due block access faults on a particular block. The thread must be blocked due to a page fault or block access fault. For page faults, the faulting instruction is reissued. For block access faults, the faulting instruction may be reissued or the access may be completed without reissuing (e.g., if the faulting access was a buffered store). The block access tag may or may not be checked again; that is, if the faulting access still conflicts with the tag value, whether the access completes or another block access fault occurs is implementation-dependent. Thus a user protocol must eventually change the access tag to make the access legal in order to achieve forward progress.

References

[1] Czarek Dubnicki and Thomas J. LeBlanc. Adjustable block size coherence caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 170–179, June 1992.

[2] Özalp Babaouglu and William Joy. Converting a swap-based system to do paging in an architecture lacking page-referenced bits. In *Proceedings of the Eighth ACM Symposium on Operating System Principles (SOSP)*, pages 78–86, December 1981.

[3] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[4] Steven K. Reinhardt, James R. Larus, and David A. Wood. Typhoon and Tempest: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[5] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–306, March 1994.

[6] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.