GASNet Specification

Version 1.6
Released August 19th, 2004
$Date: 2004/08/18 09:37:49 $
Printed 18 August 2004
$Revision: 1.34 $

Editor: Dan Bonachea bonachea@cs.berkeley.edu
http://www.cs.berkeley.edu/~bonachea/gasnet

This the GASNet specification, version 1.6.

Selected portions adapted from:

- *A. Mainwaring and D. Culler,* "*Active Message Applications Programming Interface and Communication Subsystem Organization*"*, U.C. Berkeley Computer Science Technical Report, 1996.*
- *D. Culler et al.,* "*Generic Active Message Interface Specification v1.1*"*, U.C. Berkeley Computer Science Technical Report, Feb, 1995.*

Published by LBNL FTG and U.C. Berkeley

Table of Contents

1 Introduction

1.1 Scope

This GASNet specification describes a network-independent and language-independent high-performance communication interface intended for use in implementing the runtime system for global address space languages (such as UPC or Titanium). GASNet stands for "**G**lobal-**A**ddress **S**pace **Net**working".

1.2 Organization

The interface is divided into 2 layers - the GASNet core API and the GASNet extended API:

- The extended API is a richly expressive and flexible interface that provides medium and high-level operations on remote memory and collective operations (basically anything that we could imagine being implemented using hardware support on some NIC's).
- The core API is a narrow interface based on the Active Messages paradigm, which is general enough to implement everything in the extended API.

The core API is the minimum interface that must be implemented on each network when porting to a new system, and we provide a network-independent reference implementation of the extended API which is written purely in terms of the core API to ease porting and quick prototyping. Implementors for NIC's that provide some hardware support for higher-level messaging operations (e.g. support for servicing remote reads/writes on the NIC without involving the main CPU) are encouraged to also implement an appropriate subset of the extended API directly on the network of interest (bypassing the core API) to achieve maximal performance for those operations (but this is an optimization and is not required to have a working system). Most clients will use calls to the extended API functions to implement the bulk of their communication work (thereby ensuring optimal performance across platforms). However the client is also permitted to use the core active message interface to implement non-trivial language-specific or compiler-specific communication operations which would not be appropriate in a language-independent API (e.g. implementing distributed language-level locks, distributed garbage collection, collective memory allocation, etc.).

Note the extended API interface is meant primarily as a low-level compilation target, not a library for hand-written code - as such, the goals of expressiveness and performance generally take precedence over readability and minimality.

1.3 Conventions

- All GASNet entry points are lower-case identifiers with the prefix `gasnet_`
- All constants are upper-case and preceded with the prefix `GASNET_`
- Clients access the GASNet interface by including the header '`gasnet.h`' and linking the appropriate library
- Except where otherwise noted, any of the operations in the GASNet interface could be implemented using macros or inline functions in an actual implementation - they are specified using function declaration syntax below to make the types clear, and all correct client code must type check using the definitions below. In no case should client code assume it can create a "function pointer" to any of these operations. Any macro implementations will ensure that arguments are evaluated exactly once.
- Implementation-specific values in declarations are indicated using "???"
- Sections marked "Implementor's note" are recommendations to implementors and are not part of the specification

1.4 Definitions

- **node** - An OS-level process which returns from `gasnet_init()`, and its associated local memory space and system resources. The basic unit of control when interfacing with GASNet.
- **thread** - A single thread of control within a GASNet node, which possibly shares a virtual memory space and OS-level process-id with other threads in the node. Clients which may concurrently call GASNet from more than a single thread must compile to the multi-threaded version of the GASNet library. Except where otherwise noted, GASNet makes no distinction between the threads within a multi-threaded node, and all control functions (e.g. barriers) should be executed by a single thread on the node on behalf of all local threads.
- **job** - The collection of nodes making up a parallel execution environment. Nodes often correspond to physical, architectural units, but this need not be the case (e.g. nodes may share a physical CPU/memory/NIC in multiprogrammed systems with sufficient sharable resources - note that some GASNet implementations may limit the number nodes which can run concurrently on a single system based on the number of physical network interfaces)

1.5 Configuration of GASNet

Client code must `#define` exactly one of `GASNET_PAR`, `GASNET_PARSYNC` or `GASNET_SEQ` when compiling the GASNet library and the client code (before including '`gasnet.h`') to indicate the threading environment.

`GASNET_PAR`
: The most general configuration. Indicates a fully multi-threaded and thread-safe environment - the client may call GASNet concurrently from more than one thread. The exact threading system in use is system-specific, although for obvious reasons both GASNet and the client code must agree on the threading system - unless otherwise noted, the default mechanism is POSIX threads.

`GASNET_PARSYNC`
: Indicates a multi-threaded but non-concurrent (non- threadsafe) GASNet environment, where multiple client threads may call GASNet, but their accesses to GASNet are fully serialized (e.g. by some level of synchronization above the GASNet interface). GASNet may safely assume that it will never be called from more than one client thread *concurrently* (and the client must ensure this property holds). Client code must still use GASNet No-Interrupt Sections and Handler-Safe Locks to ensure correct operation.

`GASNET_SEQ`
: Indicates a single-threaded, non-threadsafe environment. GASNet may safely assume that it will only ever be called from one unique client thread. Client code must still use GASNet No-Interrupt Sections and Handler-Safe Locks to ensure correct operation.

Implementor's Note:

- We may be able to make GASNet implementations independent of the threading system by having the client provide a few callback functions (e.g. mutex create/lock/unlock, thread create, threadid query and thread-local- data set/get)
- change the name of gasnet_init based on which mode is selected to ensure correct version is linked
- An implementation of `GASNET_PAR` is sufficient to handle all the configurations - the other configurations just permit certain useful optimizations (such as removing unnecessary locking in the library)
- Interrupt-driven implementations of `GASNET_SEQ` and `GASNET_PARSYNC` using signals must be prepared to handle the case where the thread responding to the signal may not be the thread currently inside a GASNet call. They may also need to use a private lock during HSL release to prevent multiple threads from polling simultaneously

1.6 Errors

Many GASNet core functions return 0 on success (GASNET_OK), or else they return errors from the following list, as specified by each function:

GASNET_OK = 0 (no error)

GASNET_ERR_RESOURCE

GASNET_ERR_BAD_ARG

GASNET_ERR_NOT_INIT

GASNET_ERR_BARRIER_MISMATCH

GASNET_ERR_NOT_READY

Except where otherwise noted, errors that occur during a call to the extended API are fatal.

Many of the core API functions will return GASNET_ERR_RESOURCE to indicate a generic failure in the hardware or communications system, GASNET_ERR_BAD_ARG to indicate an illegal client argument, or GASNET_ERR_NOT_INIT to indicate that gasnet_attach() has not been called.

If any node of a GASNet job crashes, aborts, or suffers a fatal hardware error, GASNet should make every attempt to ensure that the remaining nodes of the job are terminated in a timely manner to prevent creation of orphaned processes.

1.6.0.1 gasnet_ErrorName, gasnet_ErrorDesc

char * gasnet_ErrorName (int *errval*)
char * gasnet_ErrorDesc (int *errval*)

gasnet_ErrorName() and gasnet_ErrorDesc() convert the GASNet error number *errval* into a string containing the name or description (respectively) of the given error number. The client must not modify the string returned.

1.7 GASNet Types

gasnet_node_t
: unsigned integer type representing a unique 0-based node index

gasnet_handle_t
: an opaque type representing a non-blocking operation in-progress initiated using the extended API

gasnet_handler_t
: an unsigned integer type representing an index into the core API AM handler table

gasnet_handlerarg_t
: a 32-bit signed integer type which is used to express the user-provided arguments to all AM handlers. Platforms lacking a native 32-bit type may define this to a 64-bit type, but only the lower 32-bits are transmitted during an AM message send (and sign-extended on the receiver).

gasnet_token_t
: an opaque type passed to core API handlers which may be used to query message information

gasnet_register_value_t
: the largest unsigned integer type that can fit entirely in a single CPU register for the current architecture and ABI. SIZEOF_GASNET_REGISTER_VALUE_T is a preprocess-time literal integer constant (i.e. not sizeof())indicating the size of this type in bytes

gasnet_handlerentry_t
: struct type used to negotiate handler registration in gasnet_attach()

1.8 Compile-time constants

`GASNET_VERSION`
: an integer representing the major version of the GASNet spec to which this implementation complies. Implementations of this version of the specification should set this value to the integer 1

`GASNET_CONFIG_STRING`
: a string representing any the relevant GASNet compile-time configuration settings that can be compared using string compare to verify version compatibility. The string is also embedded into the library itself such that it can be scanned for within a binary executable which is statically linked with GASNet.

`GASNET_MAXNODES`
: an integer representing the maximum number of nodes supported in a single GASNet job

`GASNET_ALIGNED_SEGMENTS`
: defined by the GASNet implementation to the value 1 if `gasnet_attach()` guarantees that the remote-access memory segment will be aligned at the same virtual address on all nodes. Defined to 0 otherwise.

`GASNET_PAGESIZE`
: a preprocessor constant integer which provides the memory granularity size used for various GASNet parameters which are required to be page-aligned. On many systems this will be the system page size.

1.9 General notes

- All GASNet functions (in the extended *and* core API) support loopback (i.e. a node sending a get or active message to itself), and all functions will still work in the case of single-node jobs (e.g. barriers are basically no-ops in that case)
- GASNet will ensure that stdout/stderr are correctly propagated in a system-specific way (e.g. to the spawning console or possibly to a file or set of files). No guarantees are made about propagation of stdin, although some implementations may choose to deal with this.
- GASNet makes no guarantees about the propagation of external signals across a job - however, see comments in `gasnet_exit`

2 Core API

The core API consists of:

- A job control interface for bootstrapping, job termination and job environment queries
- The active messaging interface for implementing requests, replies and handlers
- An interface which provides handler signal-safety and atomicity control (No-Interrupt Sections and Handler-Safe Locks)

2.1 Job Control Interface

Job startup in GASNet is a two-step process. GASNet programs should start by calling gasnet_init() as the first statement in their main() function, which bootstraps the nodes and establishes command-line arguments and the job environment. All nodes then call the gasnet_attach() function to initialize the network and register shared memory segments.

GASNet initialization may register some UNIX signal handlers (e.g. to support interrupt-based implementations or aggressive segment registration policies). Client code which registers signal handlers must be careful not to preempt any GASNet-registered signal handlers (even for seemingly fatal signals such as `SIGABRT`) - the only signal which the client may always safely catch is `SIGQUIT`.

Any GASNet library implementation can be built in one of the following three configurations, which affects the behavior of remote-access memory segment registration during gasnet_attach(). The `gasnet.h` header file will define the appropriate preprocessor symbol to indicate which configuration is active.

`GASNET_SEGMENT_FAST`
: The remote-access memory segment is limited to an implementation-defined "reasonable" size, and optimized in an implementation-specific way to provide the fastest possible remote accesses. The maximum segment size may be queried using `gasnet_getMaxLocalSegmentSize()`.

`GASNET_SEGMENT_LARGE`
: This configuration allows clients with larger shared data requirements to register a larger remote-access memory segment, possibly at some cost in the efficiency of remote accesses. The maximum segment size may be queried using `gasnet_getMaxLocalSegmentSize()`, and should be comparable to the maximum total data size allowed for processes on the given system.

`GASNET_SEGMENT_EVERYTHING`
: The entire virtual memory space of each process is made available for remote access, in a way such that any memory access that would succeed when executed locally by this node would also succeed if executed by other nodes remotely. This can be used by clients which need to make the entire memory heap and static data areas available for remote access.

Implementor's Note:

- The maximum segment size for `GASNET_SEGMENT_FAST` on many implementations is likely to to be limited by factors such as the amount of pinnable physical memory currently available in the system, and the access range of the NIC hardware.
- `GASNET_SEGMENT_EVERYTHING` support can trivially be provided by implementing all the remote-access operations and long AM messages using core API medium messages, such that all data accesses are actually executed by the local host processor. However, implementors are encouraged to investigate higher-performance alternatives whenever possible.
- On systems requiring pinned segments, `GASNET_SEGMENT_LARGE` can be implemented using dynamic pinning schemes (possibly with caching to amortize rendezvous and pinning costs) or combinations of direct remote accesses and AM-based accesses.

2.1.0.1 gasnet_init

int gasnet_init (int *argc, char ***argv)

Bootstraps a GASNet job and performs any system-specific setup required.

Called by all GASNet-based applications upon startup to bootstrap the nodes, before any other processing takes place. Must be called before any calls to any other functions in this specification, and before any investigation of the command-line parameters passed to the program in *argc*/*argv*, which may be modified or augmented by this call. The semantics of any code executing before the call to `gasnet_init()` is implementation-specific (for example, it is undefined whether `stdin/stdout/stderr` are functional, or even how many nodes will run that code).

Upon return from `gasnet_init()`, all the nodes of the job will be running, stdout/stderr will be functional, and the basic job environment will be established, however the primary network resources may not yet have been initialized. The following GASNet functions are the only ones that may be called between `gasnet_init()` and `gasnet_attach()`:

```
gasnet_mynode()
gasnet_nodes()
gasnet_getMaxLocalSegmentSize()
gasnet_getMaxGlobalSegmentSize()
gasnet_getenv()
gasnet_exit()
```

All other GASNet calls are prohibited until after a successful `gasnet_attach()`.

`gasnet_init()` may fail with a fatal error and implementation-defined message if the nodes of the job cannot be successfully bootstrapped. It also may return an error code such as `GASNET_ERR_RESOURCE` to indicate there was a problem acquiring network or system resources. Otherwise, it returns `GASNET_OK` to indicate success. May only be called once during a process lifetime, subsequent calls will return an error.

2.1.0.2 gasnet_attach

```
typedef struct {
  gasnet_handler_t index; // == 0 for don't care
  void (*fnptr)();
} gasnet_handlerentry_t;
```

int gasnet_attach (gasnet_handlerentry_t *table, int numentries,
uintptr_t segsize, uintptr_t minheapoffset)

Initializes the GASNet network system and performs any system-specific setup required.

table is an array of *numentries* gasnet_handlerentry_t elements used for registering active-message handlers provided by the client code. Clients that never explicitly call the active-message request functions in the core API need not register any handlers, and may pass a `NULL` pointer for *table*. Clients wishing to register some handlers should fill in *table* with function pointers and the desired handler index (or index 0 for "don't-care") - note that handlers 0..127 are reserved for GASNet internal use, and handlers 128..255 are available for client-provided handlers. Once `gasnet_attach()` returns, any "don't care" handler indexes in the table will be modified in place to reflect the handler index assigned for each handler - the assignment algorithm is deterministic: passing the same handler table on each node will guarantee an identical resulting assignment on each node. Handler function prototypes should match the prototypes described in the Active Message Interface section.

segsize and *minheapoffset* are used to communicate the desired size and location of the remote-access memory data segment for the local node that will be used for all remote accesses (i.e. using the data transfer functions of the extended API) or as the target of any Long active-messages in the core API. The client passes the desired size of this area in bytes as *segsize*, which must be a multiple of `GASNET_PAGESIZE`, and should be less than or equal to the value returned by `gasnet_getMaxLocalSegmentSize()`. *minheapoffset* specifies the minimum amount of virtual memory space (in bytes) to leave between the end of the current memory heap and the beginning of the remote-access memory segment (on some systems the size of this offset may

limit the total future growth of the local memory heap, on other systems it may be irrelevant). All nodes are required to pass the same value for *minheapoffset*. Note that specifying a large *minheapoffset* may limit the possible size of the remote-access segment on some systems. Passing a *segsize* of zero disables the remote-access segment for this node, meaning other nodes cannot access it with remote-memory operations and this node cannot be the target of any Long AM messages.

GASNet will attempt to place the data segment in an area of the virtual memory space whose pages are currently unused (e.g. by calling `mmap`). The actual remote-access segment size achieved may be less than *segsize* if insufficient system resources are available - the exact size and location of the segment for all nodes should be queried after attach using `gasnet_getSegmentInfo()`. The segment assignment is guaranteed to have a `GASNET_PAGESIZE`-aligned base address and size, but may differ in size across nodes, according to the requested segment sizes and system resource availability. GASNet will not initialize data within the memory segment in any way, nor will it attempt to access the memory locations within the segment until directed to do so by a data transfer function or Long active message.

If the GASNet implementation defines the macro `GASNET_ALIGNED_SEGMENTS` to 1, then `gasnet_attach()` guarantees that the base of the remote-access memory segment will be aligned at the same virtual address across all nodes (and will fail if it cannot provide this). Otherwise, this guarantee is not provided. Note the segment sizes may still differ across nodes, based on *segsize* and system resource availability.

In the `GASNET_SEGMENT_FAST` and `GASNET_SEGMENT_LARGE` configurations, GASNet guarantees that data transfer functions, Long active messages and local accesses referencing memory locations in the remote-access memory segment will succeed, even before any local activity takes place on those pages (i.e. in an implementation performing lazy registration, first touch = allocate).

segsize and *minheapoffset* are ignored in the `GASNET_SEGMENT_EVERYTHING` configuration, as the entire virtual memory space is implicitly shared for remote access. Under this configuration, it is the client's responsibility to ensure that any remote-memory references fall within the legal areas of the current heap and data segment for the target node - remote accesses or Long active messages to locations outside these areas will have undefined effects (for example, they *may* cause a segmentation fault on the target node).

`gasnet_attach()` may fail with a fatal error and implementation-defined message if the network cannot be successfully initialized. It also may return an error code such as `GASNET_ERR_RESOURCE` to indicate there was a problem acquiring network or system resources. Otherwise, it returns `GASNET_OK` to indicate success.

A successful call acts as a global barrier and blocks until all other nodes which are part of this parallel job have successfully called `gasnet_attach()`. May only be called once during a process lifetime, subsequent calls will return an error.

Implementor's Note:

- In the `GASNET_SEGMENT_FAST` and `GASNET_SEGMENT_LARGE` configurations, GASNet must take steps to ensure the pages in the segment have been properly registered for remote access in a system-specific and implementation-specific way (e.g. mmapping them so they get added to the process page table, pinning the pages, registering the physical address with the NIC, etc.). Implementations are encouraged to defer consuming physical memory or swap space resources for pages in the segment until the first actual reference to them.
- Every implementation that pins pages needs a strategy for handling remote accesses under the `GASNET_SEGMENT_LARGE` and `GASNET_SEGMENT_EVERYTHING` configurations when the segment size exceeds the amount of pinnable pages - e.g. some implementations may dynamically pin pages, others may pin only a portion of the segment and use an extra copy to handle access to data outside the pinned region.
- Some GASNet implementations may need to allocate and pin additional memory for their own internal use in messaging (e.g. send buffers), but such memory should not fall within the client's data segment under `GASNET_SEGMENT_FAST` and `GASNET_SEGMENT_LARGE` (although it may be adjacent to it).
- Some GASNet implementations may also choose to pin other pages to optimize access and remove extra copies - for example, pinning the program stack may be advisable on some systems since a large number of the data transfer functions in the extended API are likely to use stack locations as the local source/destination.

2.1.0.3 gasnet_getMaxLocalSegmentSize

`uintptr_t` **gasnet_getMaxLocalSegmentSize** ()

Retrieve an approximate, optimistic maximum size in bytes for the remote-access memory segment that may be provided to `gasnet_attach()` under the current configuration.

The return value of this function may depend on current system resource usage, and may return different values on different nodes of a job, according to current system utilization. The value returned will always be a multiple of `GASNET_PAGESIZE`.

The value returned is an optimistic approximation of the segment size which can be acquired by `gasnet_attach()` - the actual size achieved can be queried after attach using `gasnet_getSegmentInfo()`.

On many implementations, this function will return different values in the `GASNET_SEGMENT_FAST` and `GASNET_SEGMENT_LARGE` configurations. Under the `GASNET_SEGMENT_EVERYTHING` configuration, this function returns -1.

This function has undefined behavior after `gasnet_attach()`.

2.1.0.4 gasnet_getMaxGlobalSegmentSize

`uintptr_t` **gasnet_getMaxGlobalSegmentSize** ()

Returns a global minimum value that would be returned by a call to `gasnet_getMaxLocalSegmentSize` on any node of the current job (i.e. the smallest max segment size estimated for any node in the job).

This function has undefined behavior after `gasnet_attach()`.

2.1.0.5 gasnet_exit

`void` **gasnet_exit** (`int` *exitcode*)

Terminate the current GASNet job and return the given *exitcode* to the console which invoked the job (in a system-specific way). This call is *not* a collective operation, meaning any node may call it at any time after initialization. It causes the system to flush all I/O, release all resources and terminate the job for all active nodes. If several nodes and/or threads call it simultaneously with different exit codes within a given synchronization phase, the result provided to the console will be one of the provided exit codes (chosen arbitrarily). This function should be called at the end of `main()` after a barrier to ensure proper system exit, and should also be called in the event of any fatal errors. GASNet clients are encouraged to call `gasnet_exit()` before explicitly exiting (by calling `exit(), abort()`) to reduce the possibility and lifetime of orphaned nodes, but this is not required.

GASNet will send a `SIGQUIT` signal to the node if it detects that a remote node has called `gasnet_exit` or crashed (in which case the node should catch the signal, perform any system-specific shutdown, then call `gasnet_exit()` to end the local node process). GASNet will also send a `SIGQUIT` signal if it detects that the job has received a different catchable terminate-the-program signal (e.g. `SIGTERM`, `SIGINT`) since some of these other signals may be meaningful (and non-fatal) to certain GASNet implementations.

2.2 Job Environment Queries

2.2.0.1 gasnet_mynode

`gasnet_node_t` **gasnet_mynode** ()

returns the unique, 0-based node index representing this node in the current GASNet job

2.2.0.2 gasnet_nodes

`gasnet_node_t` **gasnet_nodes** ()

returns the number of nodes in the current GASNet job

2.2.0.3 gasnet_getSegmentInfo

```
typedef struct {
  void *addr;
  uintptr_t size;
} gasnet_seginfo_t;
```

`int` **gasnet_getSegmentInfo** (`gasnet_seginfo_t *`*seginfo_table*`, int` *numentries*)

Query the segment base addresses and sizes for all the nodes in the job. *seginfo_table* is an array of `gasnet_seginfo_t` (and *numentries* is the number of entries in the table). GASNet fills in the table with the remote-access segment base address and size in bytes for each node whose index is less than *numentries*. The value of *numentries* is usually equal to `gasnet_nodes()`, but is permitted to be greater (in which case higher array entries are left untouched) or less (in which case the higher-numbered nodes are not reported). This is a non-collective operation. Returns `GASNET_OK` on success.

Note that when GASNET_ALIGNED_SEGMENTS=1, the base addresses are guaranteed to be equal (i.e. all remote-access segments start at the same virtual addresses). However, in any case the segment sizes may differ across nodes, and specifically they may differ from the size requested by the client in the `gasnet_attach()` size hint.

2.2.0.4 gasnet_getenv

`char * `**gasnet_getenv** (`const char *`*name*)

Has the same semantics as the POSIX `getenv()` call, except it queries the system-specific environment which was used to spawn the job (e.g. the environment of the spawning console). Calling POSIX `getenv()` directly on some implementations may not correctly return values reflecting the environment that initiated the job spawn, consequently GASNet clients wishing to query a consistent snapshot of the spawning environment across nodes should never call `getenv()` directly. The semantics of POSIX `setenv()` are undefined in GASNet jobs (specifically, it will probably fail to propagate changes across nodes).

2.3 Active Messaging Interface

Active message communication is formulated as logically matching request and reply operations. Upon receipt of a request message, a request handler is invoked; likewise, when a reply message is received, the reply handler is invoked. Request handlers can reply at most once to the requesting node. If no explicit reply is made, the layer may generate one (to an implicit do-nothing reply handler). Thus a request handler can call reply at most once, and may only reply to the requesting node. Reply handlers cannot request or reply.

Here is a high-level description of a typical active message exchange between two nodes, A and B:

1. A calls `gasnet_AMRequest*()` to send a request to B. The call includes arguments, data payload, the node index of B and the index of the request handler to run on B when the request arrives
2. At some later time, B receives the request, and runs the appropriate request handler with the arguments and data (if any) provided in the `gasnet_AMRequest*()` call. The request handler does some work on the arguments, and usually finishes by calling `gasnet_AMReply*()` to issue a reply message before it exits (replying is optional in GASNet, but required in AM2 - if the request handler does not reply then no further actions are taken). `gasnet_AMReply*()` takes the token passed to the request handler, arguments and data payload, and the index of the reply handler to run when the reply message arrives. It does not take a node index because a request handler is only permitted to send a reply to the requesting node
3. At some later time, A receives the reply message from B and runs the appropriate reply handler, with the arguments and data (if any) provided in the `gasnet_AMReply*()` call. The reply handler does some work on the arguments and then exits. It is not permitted to send further messages.

The message layer will deliver requests and replies to destination nodes barring any catastrophic errors (e.g. node crashes). From a sender's point of view, the request and reply functions block until the message is sent. A message is defined to be sent once it is safe for the caller to reuse the storage (registers or memory) containing the message (one notable exception to this policy is `gasnet_RequestLargeAsyncM()`). In implementations which

copy or buffer messages for transmission, the definition still holds: message sent means the layer has copied the message and promises to deliver the copy with its "best effort", and the original message storage may be reused. By best effort, the message layer promises it will take care of all the details necessary to transmit the message. These details include any retransmission attempts and buffering issues on unreliable networks.

However, in either case, sent does not imply received. Once control returns from a request or reply function, clients cannot assume that the message has been received and handled at the destination. The message layer only guarantees that if a request or reply is sent, and, if the receiver occasionally polls for arriving messages, then the message will eventually be received and handled. From a receiver's point of view, a message is defined to be received only once its handler function is invoked. The contents of partially received messages and messages whose handlers have not executed are undefined.

If the client sends an AM request or AM reply to a handler index which has not been registered on the destination node, GASNet will print an implementation-defined error message and terminate the job. It is implementation-defined whether this checking happens on the sending or receiving node.

2.3.1 Active Message Categories

There are three categories of active messages:

'`Short Active Message`'
: These messages carry only a few integer arguments (up to `gasnet_AMMaxArgs()`)
handler prototype:

```
void handler(gasnet_token_t token,
             gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

'`Medium Active Message`'
: In addition to integer arguments, these messages can can carry an opaque data payload (up to `gasnet_AMMaxMedium()` bytes in length), that will be made available to the handler when it is run on the remote node.
handler prototype:

```
void handler(gasnet_token_t token,
             void *buf, size_t nbytes,
             gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

'`Long Active Message`'
: In addition to integer arguments, these messages can carry an opaque data payload (up to `gasnet_AMMaxLong()` bytes in length) which is destined for a particular predetermined address in the segment of the remote node (often implemented using RDMA hardware assistance)
handler prototype:

```
void handler(gasnet_token_t token,
             void *buf, size_t nbytes,
             gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

For more discussion on these three categories, see the Appendix.

The number of handler arguments (M) is specified upon issuing a request or reply by choosing the request/reply function of the appropriate name. The category of message and value of M used in the request/reply message sends determines the appropriate handler prototype, as detailed above. If a request or reply is sent to a handler whose prototype does not match the requirements as detailed above, the result is undefined.

Implementor's Note:

- Some implementations may choose to optimize medium and long messages for payloads whose base address and length are aligned with certain convenient sizes (word-aligned, doubleword-aligned, page-aligned etc.) but this does not affect correctness.

2.3.2 Active Message Size Limits

These functions are used to query the maximum size messages of each category supported by a given implementation. These are likely to be implemented as macros for efficiency of client code which uses them (within packing loops, etc.)

2.3.2.1 gasnet_AMMaxArgs

`size_t` **gasnet_AMMaxArgs** ()

Returns the maximum number of handler arguments (i.e. M) that may be passed with any AM request or reply function. This value is guaranteed to be at least `(2 * MAX(sizeof(int),sizeof(void*)))` (i.e. 8 for 32-bit systems, 16 for 64-bit systems), which ensures that 8 ints and/or pointers can be sent with any active message. All implementations must support *all* values of M from 0...`gasnet_AMMaxArgs()`.

2.3.2.2 gasnet_AMMaxMedium

`size_t` **gasnet_AMMaxMedium** ()

Returns the maximum number of bytes that can be sent in the payload of a single medium AM request or reply. This value is guaranteed to be at least 512 bytes on any implementation.

2.3.2.3 gasnet_AMMaxLongRequest

`size_t` **gasnet_AMMaxLongRequest** ()

Returns the maximum number of bytes that can be sent in the payload of a single long AM request. This value is guaranteed to be at least 512 bytes on any implementation. Implementations which use RDMA to implement long messages are likely to support a much larger value.

2.3.2.4 gasnet_AMMaxLongReply

`size_t` **gasnet_AMMaxLongReply** ()

Returns the maximum number of bytes that can be sent in the payload of a single long AM reply. This value is guaranteed to be at least 512 bytes on any implementation. Implementations which use RDMA to implement long messages are likely to support a much larger value.

2.3.3 Active Message Request Functions

In the function descriptions below, M is to be replaced with a number in [0 ... `gasnet_AMMaxArgs()`]

2.3.3.1 gasnet_AMRequestShortM

`int` **gasnet_AMRequestShortM** (`gasnet_node_t` *dest*, `gasnet_handler_t` *handler*,
`gasnet_handlerarg_t` *arg0*, ..., `gasnet_handlerarg_t` *argM-1*);

Send a short AM request to node *dest*, to run the handler registered on the destination node at handler table index *handler*, with the given M arguments. `gasnet_AMRequestShortM` returns control to the calling thread of computation after sending the request message. Upon receipt, the receiver invokes the appropriate active message request handler function with the M integer arguments. Returns `GASNET_OK` on success.

2.3.3.2 gasnet_AMRequestMediumM

`int` **gasnet_AMRequestMediumM** (`gasnet_node_t` *dest*, `gasnet_handler_t` *handler*,
`void *`*source_addr*, `size_t` *nbytes*, `gasnet_handlerarg_t` *arg0*, ...,
`gasnet_handlerarg_t` *argM-1*)

Send a medium AM request to node *dest*, to run the handler registered on the destination node at handler table index *handler*, with the given M arguments.

The message also carries a data payload copied from the local node's memory space as indicated by *source_addr* and *nbytes* (which need not fall within the registered data segment on the local node). The value of *nbytes* must be no larger than the value returned by `gasnet_AMMaxMedium()`, and is permitted to be zero (in which case *source_addr* is ignored and the *buf* value passed to the handler is undefined).

`gasnet_AMRequestMediumM` returns control to the calling thread of computation after sending the associated request, and the source memory may be freely modified once the function returns. The active message is logically delivered after the data transfer finishes.

Upon receipt, the receiver invokes the appropriate request handler function with a pointer to temporary storage containing the data payload, the number of data bytes transferred, and the M integer arguments. The dynamic scope of the storage is the same as the dynamic scope of the handler. The data should be copied if it is needed beyond this scope. Returns `GASNET_OK` on success.

2.3.3.3 gasnet_AMRequestLongM

`int` **gasnet_AMRequestLongM** (`gasnet_node_t` *dest*, `gasnet_handler_t` *handler*,
`void *`*source_addr*, `size_t` *nbytes*, `void *`*dest_addr*, `gasnet_handlerarg_t` *arg0*, ...,
`gasnet_handlerarg_t` *argM-1*);

Send a long AM request to node *dest*, to run the handler registered on the destination node at handler table index *handler*, with the given M arguments.

The message also carries a data payload copied from the local node's memory space as indicated by *source_addr* and *nbytes* (which need not fall within the registered data segment on the local node). The value of nbytes must be no larger than the value returned by `gasnet_AMMaxLongRequest()`, and is permitted to be zero (in which case *source_addr* is ignored and the *buf* value passed to the handler is undefined). The memory specified by [*dest_addr*...(*dest_addr*+*nbytes*-1)] must fall entirely within the memory segment registered for remote access by the destination node. This area will receive the data transfer before the handler runs.

If *dest* is the current node (i.e. loopback) and the source and destination memory overlap, the result is undefined. `gasnet_AMRequestLongM` returns control to the calling thread of computation after sending the associated request, and the source memory may be freely modified once the function returns. The active message is logically delivered after the bulk transfer finishes. Upon receipt, the receiver invokes the appropriate request handler function with a pointer into the memory segment where the data was placed, the number of data bytes transferred, and the M integer arguments. Returns `GASNET_OK` on success.

2.3.3.4 gasnet_AMRequestLongAsyncM

`int` **gasnet_AMRequestLongAsyncM** (`gasnet_node_t` *dest*, `gasnet_handler_t` *handler*,
`void *`*source_addr*, `size_t` *nbytes*, `void *`*dest_addr*, `gasnet_handlerarg_t` *arg0*, ...,
`gasnet_handlerarg_t` *argM-1*);

`gasnet_AMRequestLongAsyncM()` has identical semantics to `gasnet_AMRequestLongM()`, except that the handler is required to send an AM reply and the data payload source memory must NOT be modified until this matching reply handler has begun execution. Some implementations may leverage this additional constraint to provide higher performance (e.g. by reducing extra data copying).

Implementor's Note:

- Note that unlike the AM2.0 function of similar name, this function is permitted to block temporarily if the network is unable to immediately accept the new request.

2.3.4 Active Message Reply Functions

The following active message reply functions may only be called from the context of a running active message request handler, and a reply function may be called at most once from any given request handler (it is an error to do otherwise).

2.3.4.1 gasnet_AMReplyShortM

`int` **gasnet_AMReplyShortM** (`gasnet_token_t` *token*, `gasnet_handler_t` *handler*,
`gasnet_handlerarg_t` *arg0*, ..., `gasnet_handlerarg_t` *argM-1*);

Send a short AM reply to the indicated *handler* on the requesting node (i.e. the node responsible for this particular invocation of the request handler), and include the given M arguments. `gasnet_AMReplyShortM` returns control to the calling thread of computation after sending the reply message.

Upon receipt, the receiver invokes the appropriate active message reply handler function with the M integer arguments. Returns `GASNET_OK` on success.

2.3.4.2 gasnet_AMReplyMediumM

`int` **gasnet_AMReplyMediumM** (`gasnet_token_t` *token*, `gasnet_handler_t` *handler*,
`void *`*source_addr*, `size_t` *nbytes*, `gasnet_handlerarg_t` *arg0*, ...,
`gasnet_handlerarg_t` *argM-1*);

Send a medium AM reply to the indicated *handler* on the requesting node (i.e. the node responsible for this particular invocation of the request handler), with the given M arguments and given data payload copied from the local node's memory space (*source_addr* need not fall within the registered data segment on the local node). The value of *nbytes* must be no larger than the value returned by `gasnet_AMMaxMedium()`, and is permitted to be zero (in which case *source_addr* is ignored and the *buf* value passed to the handler is undefined). `gasnet_AMReplyMediumM` returns control to the calling thread of computation after sending the associated reply, and the source memory may be freely modified once the function returns. The active message is logically delivered after the data transfer finishes.

Upon receipt, the receiver invokes the appropriate reply handler function with a pointer to temporary storage containing the data payload, the number of data bytes transferred, and the M integer arguments. The dynamic scope of the storage is the same as the dynamic scope of the handler. The data should be copied if it is needed beyond this scope. Returns `GASNET_OK` on success.

2.3.4.3 gasnet_AMReplyLongM

`int` **gasnet_AMReplyLongM** (`gasnet_token_t` *token*, `gasnet_handler_t` *handler*,
`void *`*source_addr*, `size_t` *nbytes*, `void *`*dest_addr*, `gasnet_handlerarg_t` *arg0*, ...,
`gasnet_handlerarg_t` *argM-1*);

Send a long AM reply to the indicated *handler* on the requesting node (i.e. the node responsible for this particular invocation of the request handler), with the given M arguments and given data payload copied from the local node's memory space (*source_addr* need not fall within the registered data segment on the local node). The value of *nbytes* must be no larger than the value returned by `gasnet_AMMaxLongReply()`, and is permitted to be zero (in which case *source_addr* is ignored and the *buf* value passed to the handler is undefined). The memory specified by [*dest_addr*...(*dest_addr*+*nbytes*-1)] must fall entirely within the memory segment registered for remote access by the destination node. If dest is the current node (i.e. loopback) and the source and destination memory overlap, the result is undefined. `gasnet_AMReplyLongM` returns control to the calling thread of computation after sending the associated reply, and the source memory may be freely modified once the function returns. The active message is logically delivered after the bulk transfer finishes.

Upon receipt, the receiver invokes the appropriate reply handler function with a pointer into the memory segment where the data was placed, the number of data bytes transferred, and the M integer arguments. Returns `GASNET_OK` on success.

2.3.5 Misc. Active Message Functions

2.3.5.1 gasnet_AMPoll

`int` **gasnet_AMPoll** ()

An explicit call to service the network, process pending messages and run handlers as appropriate. Most of the message-sending primitives in GASNet poll the network implicitly. Purely polling-based implementations of GASNet may require occasional calls to this function to ensure progress of remote nodes during compute-only loops. Any client code which spin-waits for the arrival of a message should call this function within the spin loop to optimize response time. This call may be a no-op on some implementations (e.g. purely interrupt-based implementations). Returns `GASNET_OK` unless an error condition was detected.

2.3.5.2 GASNET_BLOCKUNTIL

```
#define GASNET_BLOCKUNTIL(cond) ???
```

This is a macro which implements a busy-wait/blocking polling loop in the way most efficient for the current GASNet core implementation. The macro blocks execution of the current thread and services the network until the provided condition becomes true. *cond* is an arbitrary C expression which will be evaluated by the macro one or more times as active messages arrive until the condition evaluates to a non-zero value. *cond* is an expression whose value is altered by the execution of an AM handler which the client thread is waiting for - GASNet may safely assume that the value of *cond* will only change while an AM handler is executing.

Example usage:

```
int doneflag = 0;
gasnet_AMRequestShort1(..., &doneflag); // reply handler sets doneflag to 1
GASNET_BLOCKUNTIL(doneflag == 1);
```

Note that code like this would be illegal and could cause node 0 to sleep forever:

```
static int doneflag = 0;
node 0:                                  node 1:
GASNET_BLOCKUNTIL(doneflag == 1);        gasnet_put_val(0, &doneflag, 1, sizeof(int));
```

because `gasnet_put_val` (and other extended API functions) might not be implemented using AM handlers. Also note that *cond* may be evaluated concurrently with handler execution, so the client is responsible for negotiating any atomicity concerns between the cond expression and handlers (for example, protecting both with a handler-safe lock if the *cond* expression reads two or more values which are all updated by handlers). Finally, note that unsynchronized handler code which modifies one or more locations and then performs a flag write to signal a different thread may need to execute a local memory barrier before the flag write to ensure correct ordering on non-sequentially-consistent SMP hardware.

Implementor's Note:

- one trivial implementation: `#define GASNET_BLOCKUNTIL(cond) while (!(cond)) gasnet_AMPoll()`
- smarter implementations may choose to spin for awhile and then block
- Any implementation that includes blocking must ensure progress if all client threads call GASNET_BLOCKUNTIL(), and must ensure the blocked thread is awakened even if the handler is run synchronously during a `gasnet_AMPoll()` call from a different client thread. Other client threads performing sends or polls must not be prevented from making progress by the blocking thread (possibly a motivation *against* the "trivial implementation" above).

2.3.5.3 gasnet_AMGetMsgSource

`int` **gasnet_AMGetMsgSource** (`gasnet_token_t` *token*, `gasnet_node_t *`*srcindex*)

Can be called by handlers to query the source of the message being handled. The *token* argument must be the token passed into the handler on entry. Returns `GASNET_OK` on success.

2.4 Atomicity Control

2.4.1 Atomicity semantics of handlers

Handlers may run asynchronously with respect to the main computation (in an implementation which uses interrupts to run some or all handlers), and they may run concurrently with each other on separate threads (e.g. in a CLUMP implementation where several threads may be polling the network at once). An implementation using interrupts may result in handler code running within a signal handler context. Some implementations may even choose to run handlers on a separate private thread created by GASNet (making handlers asynchronous with respect to all client threads). Note that polling-based GASNet implementations are likely to poll (and possibly run handlers) from within *any* GASNet call (i.e. not just `gasnet_AMPoll()`). Because of all this, handler code should run quickly and to completion without making blocking calls, and should not make assumptions about the context in which it is being run (special care must be taken to ensure safety in a signal handler context, see below).

Regardless, handlers themselves are not interruptible - any given thread will only be running a single AM handler at a time and will never be interrupted to run another AM handler (there is one exception to this rule - the `gasnet_AMReply*()` call in a request handler may cause reply handlers to run synchronously, which may be necessary to avoid deadlock in some implementations. This should not be a problem since `gasnet_AMReply*()` is often the last action taken by a request handler). Handlers are specifically prohibited from initiating random network communication to prevent deadlock - request handlers must generate at most one reply (to the requestor) and make no other communication calls (including polling), and reply handlers may not communicate or poll at all.

The asynchronous nature of handlers requires two mechanisms to make them safe: a mechanism to ensure signal safety for GASNet implementations using interrupt-based mechanisms, and a locking mechanism to allow atomic updates from handlers to data structures shared with the client threads and other handlers.

(see http://www.cs.berkeley.edu/~bonachea/upc/ for a more detailed discussion on handler atomicity)

2.4.2 No-Interrupt Sections - Ensuring signal-safety for handlers

Traditionally, code running in signal handler context is extremely circumscribed in what it can do: e.g. none of the standard pthreads/System V synchronization calls are on the list of signal-safe functions (for such a list see *POSIX System Interfaces 2.4, IEEE Std 1003.1-2001*). Note that even most "thread-safe" libraries will break or deadlock if called from a signal handler by the same thread currently executing a different call to that library in an earlier stack frame. One specific case where this is likely to arise in practice is calls to `malloc()`/`free()`. To overcome these limitations, and allow our handlers to be more useful, the normal limitations on signal handlers will be avoided by allowing the client thread to temporarily disable the network interrupts that run handlers. All function calls that are not signal-safe and could possibly access state shared by functions also called from handlers MUST be called within a GASNet "No-Interrupt Section":

2.4.2.1 gasnet_hold_interrupts, gasnet_resume_interrupts

`void` **gasnet_hold_interrupts** ()
`void` **gasnet_resume_interrupts** ()

`gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` are used to define a GASNet No-Interrupt Section (any code which dynamically executes between the hold and resume calls is said to be "inside" the No-Interrupt Section). These are likely to be implemented as macros and highly tuned for efficiency. The hold and resume calls must be paired, and may *not* be nested recursively or the results are undefined (this means that clients should be especially careful when calling other functions in the client from within a No-Interrupt Section). Both calls will return immediately in the common case, although one or both may cause messages to be serviced on some implementations. GASNet guarantees that no handlers will run asynchronously **on the current thread** within the No-Interrupt Section. The no-interrupt state is a per-thread setting, and GASNet may continue running handlers synchronously or asynchronously on other client threads or GASNet-private threads (even in a `GASNET_SEQ` configuration) - specifically, a No-Interrupt Section does **not** guarantee atomicity with respect to handler code, it merely provides a way to ensure that handlers won't run on a given thread while it's inside a call to a non-signal-safe library.

2.4.3 Restrictions on No-Interrupt Sections

There is a strict set of conventions governing the use of No-Interrupt Sections which must be followed in order to ensure correct operation on all GASNet implementations. Clients which violate any of these rules may be subject to intermittent crashes, fatal errors or network deadlocks.

- `gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` need not be called from within a handler context - handlers are run within an implicit No-Interrupt Section, and `gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` calls are ignored within a handler context.
- Code in a No-Interrupt Section must not call any GASNet functions that may send requests or synchronously run handlers - specifically, the only GASNet functions which may legally be called within the No-Interrupt Section are:

  `gasnet_mynode(), gasnet_nodes(), gasnet_hsl_*(), gasnet_exit(), gasnet_AMReply*()`

  Note that due to the previous rule, these are also the only GASNet functions that may legally be called within a handler context (and `gasnet_AMReply*()` is only legal in a request handler).
- Code in a No-Interrupt Section must never block or spin-wait for an unbounded amount of time, especially when awaiting a result produced by a handler. The *only* exception to this rule is that a thread may call `gasnet_hsl_lock` within a No-Interrupt Section (subject to the rules in section see Section 2.4.5 [Restrictions on Handler-Safe Locks], page 17).
- No-Interrupt Sections should only be held "briefly" to avoid starving the network (could cause performance degradation, but should not affect correctness). Very long No-Interrupt Sections (i.e. on the order of 10 sec or more) could cause some GASNet implementations employing timeout-based mechanisms to fail (e.g. remote nodes may decide this node is dead and abort the job).

Implementor's Note:

- One possible implementation: Keep a bit for each thread indicating whether or not a No-Interrupt Section is in effect, which is checked by all asynchronous signal handlers. If a signal arrives while a No-Interrupt Section is in effect, a different per-thread bit in memory will be marked indicating a "missed GASNet signal": the `gasnet_resume_interrupts()` call will check this bit, and if it is set, the action for the signal will be taken (the action for a GASNet signal is always to check the queue of incoming network messages, so there's no ambiguity on what the signal meant. Since messages are queued, the single 'signal missed' bit is sufficient for an arbitrary number of missed signals during a single No-Interrupt Section - GASNet messages will be removed and processed until the queue is empty).
- Implementation needs to hold a No-Interrupt Section over a thread while running handlers or holding HSL's
- Strictly polling-based implementations which never interrupt a thread can implement these as a no-op.

2.4.4 Handler-Safe Locks

In order to support handlers atomically updating data structures accessed by the main-line client code and other handlers, GASNet provides the Handler-Safe Lock (HSL) mechanism. As the name implies, these are a special kind of lock which are distinguished as being the **only** type of lock which may be safely acquired from a handler context. There is also a set of restrictions on their usage which allows this to be safe (see below). All lock-protected data structures in the client that need to be accessed by handlers should be protected using a Handler-Safe Lock (i.e. instead of a standard POSIX mutex).

2.4.4.1 gasnet_hsl_t

`gasnet_hsl_t` is an opaque type representing a Handler-Safe Lock. HSL's operate analogously to POSIX mutexes, in that they are always manipulated using a pointer.

2.4.4.2 gasnet_hsl_init, gasnet_hsl_destroy

```
gasnet_hsl_t hsl = GASNET_HSL_INITIALIZER;
```

`void` **gasnet_hsl_init** (`gasnet_hsl_t *`*hsl*)
`void` **gasnet_hsl_destroy** (`gasnet_hsl_t *`*hsl*)

Similarly to POSIX mutexes, HSL's can be created in two ways. They can be statically declared and initialized using the `GASNET_HSL_INITIALIZER` constant. Alternately, HSL's allocated using other means (such as dynamic allocation) may be initialized by calling `gasnet_hsl_init()`. `gasnet_hsl_destroy()` may be called on either type of HSL once it's no longer needed to release any system resources associated with it. It is erroneous to call `gasnet_hsl_init()` on a given HSL more than once. It is erroneous to destroy an HSL which is currently locked. Any errors detected in HSL initialization/destruction are fatal.

2.4.4.3 gasnet_hsl_lock, gasnet_hsl_unlock

`void` **gasnet_hsl_lock** (`gasnet_hsl_t *`*hsl*)
`int` **gasnet_hsl_trylock** (`gasnet_hsl_t *`*hsl*)
`void` **gasnet_hsl_unlock** (`gasnet_hsl_t *`*hsl*)

Lock and unlock HSL's.

`gasnet_hsl_lock(hsl)` will block until the *hsl* lock can be acquired by the current thread. `gasnet_hsl_lock()` may be called from within main-line client code or from within handlers - this is the **only** blocking call which is permitted to execute within a GASNet handler context (e.g. it is erroneous to call POSIX mutex locking functions).

`gasnet_hsl_trylock(hsl)` attempts to acquire `hsl` for the current thread, returning immediately (without blocking). If the lock was successfully acquired, this function returns `GASNET_OK`. If the lock could not be acquired (e.g it was found to be held by another thread) then this function returns GASNET_ERR_NOT_READY and the lock is not acquired. It is *not* legal for an AM handler to spin-poll a lock without bound using `gasnet_hsl_trylock()` waiting for success - AM handlers must always use `gasnet_hsl_lock()` when they wish to block to acquire an HSL.

`gasnet_hsl_unlock(hsl)` releases the *hsl* lock previously acquired using `gasnet_hsl_lock(hsl)` or a successful `gasnet_hsl_trylock(hsl)`, and not yet released. It is erroneous to call any of these functions on HSL's which have not been properly initialized.

Note that under the `GASNET_SEQ` configuration, HSL locking functions may only be called from handlers and the designated GASNet client thread (*not* from other client threads that may happen to exist - those threads are not permitted to make *any* GASNet calls, which includes HSL locking calls).

All HSL locking/unlocking calls must follow the usage rules documented in the next section.

2.4.5 Restrictions on Handler-Safe Locks

There is a strict set of conventions governing the use of HSL's which must be followed in order to ensure correct operation on all GASNet implementations. Amongst other things, the restrictions are designed to ensure that HSL's are always held for a strictly bounded amount of time, to ensure that acquiring them from within a handler can't lead to deadlock. Clients which violate any of these rules may be subject to intermittent crashes, fatal errors or network deadlocks.

- Code executing on a thread holding an HSL is implicitly within a No-Interrupt Section, and must follow all the restrictions on code within a No-Interrupt Section (see Section 2.4.3 [Restrictions on No-Interrupt Sections], page 16). Calls to `gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` are ignored while holding an HSL.
- Any handler which locks one or more HSL's **must** unlock them all before returning or calling `gasnet_AMReply*()`
- HSL's may **not** be locked recursively (i.e. calling `gasnet_hsl_lock()` or `gasnet_hsl_trylock(hsl)` on a lock already held by the current thread) and attempting to do so will lead to undefined behavior. It **is** permitted for a thread to acquire more than one HSL, although the traditional cautions about the possibility of deadlock in the presence of multiple locks apply (e.g. the common solution is to define a total order on locks and always acquire them in a monotonically ascending sequence).

- HSL's must be unlocked in the reverse order they were locked (e.g. lock A; lock B; ... unlock B; unlock A; is legal - reversing the order of unlocks is erroneous)
- HSL's may not be shared across GASNet processes executing on a machine - for example, it is specifically disallowed to place an HSL in a system V or mmapped shared memory segment and attempt to access it from two different GASNet processes.

Implementor's Note:

- HSL's are likely to just be a thin wrapper around a POSIX mutex - need to add just enough state/code to ensure the safety properties (must be a real lock, even under `GASNET_PARSYNC` because client may still have multiple threads). The only specific action required is that a No-Interrupt Section is enforced while the main-line code is holding an HSL (must be careful this works properly when multiple HSL's are held or when running in a handler).
- Robust implementations may add extra error checking to help discover violations of the restrictions, at least when compiled in a debugging mode - for example, it should be easy to detect: attempts at recursive locking on HSL's, incorrectly ordered unlocks, handlers that fail to release HSL's, explicit calls to `gasnet_hold_interrupts()` and `gasnet_resume_interrupts()` in a handler or while an HSL is held or in a No-Interrupt Section, and illegal calls to GASNet messaging functions while holding an HSL or inside a No-Interrupt Section.

3 Extended API

Errors in calls to the extended API are considered fatal and abort the job (by sending a `SIGABORT` signal) after printing an appropriate error message.

3.1 Memory-to-memory Data Transfer Functions

These comments apply to all put/get functions:

- The *nbytes* parameter should be a compile-time constant whenever possible (for efficiency)
- The source memory address for all gets and the target memory address for all puts must fall within the memory area registered for remote access by the remote node (see `gasnet_attach()`), or the results are undefined
- Pointers to remote memory are passed as an ordered pair of arguments: an integer node rank (a `gasnet_node_t`) and a `void *` virtual memory address, which logically represent a global pointer to the given address on the given node. These global pointers need not be remote - the node rank passed to these functions may in fact be the rank of the current node - implementations must support this form of loopback, and should probably attempt to optimize it by avoiding network traffic for such purely local operations.
- If the source memory and destination memory regions overlap (but do not exactly coincide) the resulting value is undefined

3.2 Blocking memory-to-memory Transfers

3.2.0.1 gasnet_get, gasnet_put

`void` **gasnet_get** (`void *`*dest*`, gasnet_node_t` *node*`, void *`*src*`, size_t` *nbytes*)
`void` **gasnet_put** (`gasnet_node_t` *node*`, void *`*dest*`, void *`*src*`, size_t` *nbytes*)
Blocking get/put operations for aligned data. The get operation fetches *nbytes* bytes from the address *src* on node *node* and places them at *dest* in the local memory space. The put operation sends *nbytes* bytes from the address *src* in the local address space, and places them at the address *dest* in the memory space of node *node*. A call to these functions blocks until the transfer is complete, and the contents of the destination memory are undefined until it completes. If the contents of the source memory change while the operation is in progress the result will be implementation-specific. The *src* and *dest* addresses (whether local or remote) must be properly aligned for accessing objects of size *nbytes*. *nbytes* must be >= 0 and has no maximum size, but implementations will likely optimize for small powers of 2.

3.2.0.2 gasnet_get_bulk, gasnet_put_bulk

`void` **gasnet_get_bulk** (`void *`*dest*`, gasnet_node_t` *node*`, void *`*src*`, size_t` *nbytes*)
`void` **gasnet_put_bulk** (`gasnet_node_t` *node*`, void *`*dest*`, void *`*src*`, size_t` *nbytes*)
Blocking get/put operations for bulk (unaligned) data. These function similarly to the aligned get/put operations above, except the data is permitted to be unaligned, and implementations are likely to optimize for larger sizes of nbytes.

3.2.0.3 gasnet_memset

`void` **gasnet_memset** (`gasnet_node_t` *node*`, void *`*dest*`, int` *val*`, size_t` *nbytes*)
Blocking operation that has the same effect as if the *dest* node had executed the POSIX call `memset(`*dest*`,` *val*`,` *nbytes*`)`. As with puts, the destination memory must fall entirely within the memory area registered for remote access by the *dest* node (see `gasnet_attach`).

3.3 Non-blocking memory-to-memory transfers

The following functions provide non-blocking, split-phase memory access to shared data.

All such non-blocking operations require an initiation (generally a put or get) and a subsequent synchronization on the completion of that operation before the result is guaranteed.

There are two basic categories of non-blocking operations, defined by the synchronization mechanism used:

"*explicit handle*" *(nb) operations*
: These operations return a specific handle from the initiation that is used for synchronization. The handle can be used to synchronize a specific subset of the nb operations in-flight

"*implicit handle*" *(nbi) operations*
: These operations don't return a handle from the initiation - synchronization is accomplished by calling a synchronization routine that synchronizes all outstanding nbi operations.

3.3.1 Synchronization semantics of non-blocking data transfers

Successful synchronization of a non-blocking get operation means the local result is ready to be examined, and will contain a value held by the source location at some time in the interval between the call to the initiation function and the successful completion of the synchronization (note this specifically allows implementations to delay the underlying read until the synchronization operation is called, provided they preserve the blocking semantics of the synchronization function).

Successful synchronization of a put operation means the source data has been written to the destination location and get operations issued subsequently by any thread (or load instructions issued by the destination node) will receive the new value or a subsequently written value (assuming no other threads are writing the location)

Note that the order in which non-blocking operations complete is intentionally unspecified - the system is free to coalesce and/or reorder non-blocking operations with respect to other blocking or non-blocking operations, or operations initiated from a separate thread - the only ordering constraints that must be satisfied are those explicitly enforced using the synchronization functions (i.e. the non-blocking operation is only guaranteed to occur somewhere in the interval between initiation and successful synchronization on that operation).

Implementors should attempt to make the non-blocking initiation operations return as quickly as possible - however in some cases (e.g. when a large number of non-blocking operations have been issued or the network is otherwise busy) it may be necessary to block temporarily while waiting for the network to become available. In any case, all implementations must support at least $2^{16} - 1$ non-blocking operations in-progress - that is, the client is free to issue up to $2^{16} - 1$ non-blocking operations before issuing a sync operation, and the implementation must handle this correctly without deadlock or livelock.

3.3.2 Non-blocking memory-to-memory transfers (explicit handle)

The explicit-handle non-blocking data transfer functions return a `gasnet_handle_t` value to represent the non-blocking operation in flight. `gasnet_handle_t` is an opaque scalar type whose contents are implementation-defined, with one exception - every implementation must provide a scalar value corresponding to an "invalid" handle (`GASNET_INVALID_HANDLE`) and furthermore this value must be the result of setting all the bytes in the `gasnet_handle_t` datatype to zero. Implementators are free to define the `gasnet_handle_t` type to be any reasonable and appropriate size, although they are recommended to use a type which fits within a single standard register on the target architecture. In any case, the datatype should be wide enough to express at least $2^{16} - 1$ different handle values, to prevent limiting the number of non-blocking operations in progress due to the number of handles available. It **is** legal for clients to pass `gasnet_handle_t` values into function callees or back to function callers.

In the case of multithreaded clients (`GASNET_PAR` or `GASNET_PARSYNC`), `gasnet_handle_t` values are thread-specific. In other words, it is an error to obtain a handle value by initiating a non-blocking operation on one thread, and later pass that handle into a synchronization function from a different thread.

Any explicit-handle, non-blocking operation may return GASNET_INVALID_HANDLE to indicate it was possible to complete the operation immediately without blocking (e.g. operations where the "remote" node is actually the local node)

It is always an error to discard the gasnet_handle_t value for an explicit-handle operation in-flight - i.e. to initiate an operation and never synchronize on its completion.

3.3.2.1 gasnet_get_nb, gasnet_put_nb

gasnet_handle_t **gasnet_get_nb** (void **dest*, gasnet_node_t *node*, void **src*, size_t *nbytes*)
gasnet_handle_t **gasnet_put_nb** (gasnet_node_t *node*, void **dest*, void **src*, size_t *nbytes*)

Non-blocking get/put functions for aligned data. These functions operate similarly to their blocking counterparts, except they initiate a non-blocking operation and return immediately with a handle (gasnet_handle_t) which must later be used (by calling an explicit gasnet_*_syncnb*() function), to synchronize on completion of the non-blocking operation. The contents of the destination memory address are undefined until a synchronization completes successfully for the non-blocking operation. For the put version, the source memory may be safely overwritten once the initiation function returns.

3.3.2.2 gasnet_get_nb_bulk, gasnet_put_nb_bulk

gasnet_handle_t **gasnet_get_nb_bulk** (void **dest*, gasnet_node_t *node*, void **src*, size_t *nbytes*)
gasnet_handle_t **gasnet_put_nb_bulk** (gasnet_node_t *node*, void **dest*, void **src*, size_t *nbytes*)

Non-blocking get/put functions for bulk (unaligned) data. For the put version, the source memory may **not** be safely overwritten until a successful synchronization for the operation. If the contents of the source memory change while the operation is in progress the result will be implementation-specific. These otherwise behave identically to the non-bulk variants (but are likely to be optimized for large transfers).

3.3.2.3 gasnet_memset_nb

gasnet_handle_t **gasnet_memset_nb** (gasnet_node_t *node*, void **dest*, int *val*, size_t *nbytes*)

Non-blocking operation that has the same effect as if the *dest* node had executed the POSIX call memset(*dest*, *val*, *nbytes*). As with puts, the destination memory must fall entirely within the memory area registered for remote access by the *dest* node (see gasnet_attach).

The synchronization behavior is identical to a non-blocking, explicit-handle put operation (the gasnet_handle_t return value must be synchronized using an explicit-handle synchronization operation).

3.3.3 Synchronization for explicit-handle non-blocking operations:

GASNet supports two basic types of synchronization for non-blocking operations - trying (polling) and waiting (blocking). All explicit-handle synchronization functions take one or more gasnet_handle_t values as input and either return an indication of whether the operation has completed or block until it completes.

3.3.3.1 gasnet_wait_syncnb, gasnet_try_syncnb

void **gasnet_wait_syncnb** (gasnet_handle_t *handle*)
int **gasnet_try_syncnb** (gasnet_handle_t *handle*)

Synchronize on the completion of a single specified explicit-handle non-blocking operation that was initiated by the calling thread. gasnet_wait_syncnb() blocks until the specified operation has completed (or returns immediately if it has already completed). In any case, the handle value is "dead" after gasnet_wait_syncnb() returns and may not be used in future synchronization operations. gasnet_try_syncnb() always returns immediately, with the value GASNET_OK if the operation is complete (at which point the handle value

is "dead", and may not be used in future synchronization operations), or GASNET_ERR_NOT_READY if the operation is not yet complete and future synchronization is necessary to complete this operation.

It is legal to pass GASNET_INVALID_HANDLE as input to these functions - gasnet_wait_sync(GASNET_INVALID_HANDLE) returns immediately and gasnet_try_sync(GASNET_INVALID_HANDLE) returns GASNET_OK.

It is an error to pass a gasnet_handle_t value for an operation which has already been successfully synchronized using one of the explicit-handle synchronization functions.

3.3.3.2 gasnet_wait_syncnb_all, gasnet_try_syncnb_all

void **gasnet_wait_syncnb_all** (gasnet_handle_t *_handles_, size_t _numhandles_)
int **gasnet_try_syncnb_all** (gasnet_handle_t *_handles_, size_t _numhandles_)

Synchronize on the completion of an array of non-blocking explicit-handle operations (all of which were initiated by this thread). *numhandles* specifies the number of handles in the provided array of handles. gasnet_wait_syncnb_all() blocks until all the specified operations have completed (or returns immediately if they have all already completed). gasnet_try_syncnb_all always returns immediately, with the value GASNET_OK if all the specified operations have completed, or GASNET_ERR_NOT_READY if one or more of the operations is not yet complete and future synchronization is necessary to complete some of the operations.

Both functions will modify the provided array to reflect completions - handles whose operations have completed are overwritten with the value GASNET_INVALID_HANDLE, and the client may test against this value when gasnet_try_syncnb_all() returns GASNET_ERR_NOT_READY to determine which operations are complete and which are still pending.

It is legal to pass the value GASNET_INVALID_HANDLE in some of the array entries, and both functions will ignore it so that it has no effect on behavior. For example, if all entries in the array are GASNET_INVALID_HANDLE (or *numhandles*==0), then gasnet_try_sync_all_list() will return GASNET_OK.

3.3.3.3 gasnet_wait_syncnb_some, gasnet_try_syncnb_some

void **gasnet_wait_syncnb_some** (gasnet_handle_t *_handles_, size_t _numhandles_)
int **gasnet_try_syncnb_some** (gasnet_handle_t *_handles_, size_t _numhandles_)

These operate analogously to the gasnet_*_syncnb_all variants, except they only wait/test for at least one operation corresponding to a *valid* handle in the provided list to be complete (the valid handles values are all those which are not GASNET_INVALID_HANDLE). Specifically, gasnet_wait_syncnb_some() will block until at least one of the valid handles in the list has completed, and indicate the operations that have completed by setting the corresponding handles to the value GASNET_INVALID_HANDLE. Similarly, gasnet_try_syncnb_some will check if at least one valid handle in the list has completed (setting those completed handles to GASNET_INVALID_HANDLE) and return GASNET_OK if it detected at least one completion or GASNET_ERR_NOT_READY otherwise.

Both functions ignore GASNET_INVALID_HANDLE values so those values have no effect on behavior. If the input array is empty or consists only of GASNET_INVALID_HANDLE values, gasnet_wait_sync_some_list will return immediately and gasnet_try_sync_some_list will return GASNET_OK.

3.3.4 Non-blocking memory-to-memory transfers (implicit handle)

3.3.4.1 gasnet_get_nbi, gasnet_put_nbi, gasnet_get_nbi_bulk, gasnet_put_nbi_bulk, gasnet_memset_nbi

`void` **gasnet_get_nbi** (`void *`*dest*, `gasnet_node_t` *node*, `void *`*src*, `size_t` *nbytes*)
`void` **gasnet_put_nbi** (`gasnet_node_t` *node*, `void *`*dest*, `void *`*src*, `size_t` *nbytes*)
`void` **gasnet_get_nbi_bulk** (`void *`*dest*, `gasnet_node_t` *node*, `void *`*src*, `size_t` *nbytes*)
`void` **gasnet_put_nbi_bulk** (`gasnet_node_t` *node*, `void *`*dest*, `void *`*src*, `size_t` *nbytes*)
`void` **gasnet_memset_nbi** (`gasnet_node_t` *node*, `void *`*dest*, `int` *val*, `size_t` *nbytes*)

Non-blocking get/put functions for aligned and unaligned (bulk) data. These functions operate similarly to their explicit-handle counterparts, except they do not return a handle and must be synchronized using the implicit-handle synchronization operations. The contents of the destination memory address are undefined until a synchronization completes successfully for the non-blocking operation. As with the explicit-handle variants, the source memory for the non-bulk put operation may be safely overwritten once the initiation function returns, but the bulk put version requires the source memory to remain unchanged until the operation has been successfully completed using a synchronization.

`gasnet_memset_nbi` behaves identically to `gasnet_memset_nb`, except that it is synchronized as if it were a non-blocking, implicit-handle put operation.

3.3.5 Synchronization for implicit-handle non-blocking operations:

The following functions are used to synchronize implicit-handle non-blocking operations.

In the case of multithreaded clients, implicit-handle synchronization functions only synchronize the implicit-handle non-blocking operations initiated from the calling thread. Operations initiated by other threads sharing the GASNet interface proceed independently and are not synchronized. Implicit-handle synchronization functions will synchronize operations initiated within other function frames by the calling thread (but this cannot affect the correctness of correctly synchronized code).

3.3.5.1 gasnet_wait_syncnbi_gets, gasnet_wait_syncnbi_puts, gasnet_wait_syncnbi_all, gasnet_try_syncnbi_gets, gasnet_try_syncnbi_puts, gasnet_try_syncnbi_all

`void` **gasnet_wait_syncnbi_gets** ()
`void` **gasnet_wait_syncnbi_puts** ()
`void` **gasnet_wait_syncnbi_all** ()
`int` **gasnet_try_syncnbi_gets** ()
`int` **gasnet_try_syncnbi_puts** ()
`int` **gasnet_try_syncnbi_all** ()

These functions implicitly specify a set of non-blocking operations on which to synchronize. They synchronize on a set of outstanding non-blocking implicit-handle operations initiated by this thread - either all such gets, all such puts, or all such puts and gets (where outstanding is defined as all those implicit-handle operations which have been initiated (outside an access region) but not yet completed through a successful implicit synchronization). The wait variants block until all operations in this implicit set have completed (indicating these operations have been successfully synchronized). The try variants test whether all operations in the implicit set have completed, and return `GASNET_OK` if so (which indicates these operations have been successfully synchronized) or `GASNET_ERR_NOT_READY` otherwise (in which case *none* of these operations may be considered successfully synchronized).

If there are no outstanding implicit-handle operations, these synchronization functions all return immediately (with `GASNET_OK` for the try variants).

Implementor's Note:

- Some implementations may choose to synchronize operations from other independent threads as well, but they must ensure progress for the calling thread in the presence of another thread which is continuously initiating implicit-handle non-blocking operations.

3.3.6 Implicit access region synchronization

In some cases, it may be useful or desirable to initiate a number of non-blocking shared-memory operations (possibly without knowing how many at compile-time) and synchronize them at a later time using a single, fast synchronization. Simple implicit handle synchronization may not be appropriate for this situation if there are intervening implicit accesses which are not to be synchronized. This situation could be handled using explicit-handle non-blocking operations and a list synchronization (e.g. `gasnet_wait_syncnb_all()`), but this may not be desirable because it requires managing an array of handles (which could have negative cache effects on performance, or could be expensive to allocate when the size is not known until runtime). To handle these cases, we provide "implicit access region" synchronization, described below.

3.3.6.1 gasnet_begin_nbi_accessregion, gasnet_end_nbi_accessregion

`void` **gasnet_begin_nbi_accessregion** ();
`gasnet_handle_t` **gasnet_end_nbi_accessregion** ();

`gasnet_begin_nbi_accessregion()` and `gasnet_end_nbi_accessregion()` are used to define an implicit access region (any code which dynamically executes between the begin and end calls is said to be "inside" the region) The begin and end calls must be paired, and may not be nested recursively or the results are undefined. It is erroneous to call any implicit-handle synchronization function within the access region. All implicit-handle non-blocking operations initiated inside the region become "associated" with the abstract access region handle being constructed. `gasnet_end_nbi_accessregion()` returns an explicit handle which collectively represents all the associated implicit-handle operations (those initiated within the access region). This handle can then be passed to the regular explicit-handle synchronization functions, and will be successfully synchronized when *all* of the associated non-blocking operations (both puts and gets) initiated in the access region have completed. The associated operations cease to be implicit-handle operations, and are *not* synchronized by subsequent calls to the implicit-handle synchronization functions occurring after the access region (e.g. `gasnet_wait_syncnbi_all()`). Explicit-handle operations initiated within the access region operate as usual and do *not* become associated with the access region.

Sample code:

```
gasnet_begin_nbi_accessregion(); // begin the access region

gasnet_put_nbi_shared(...); // becomes assoc. with access region
while (...) {
  gasnet_put_nbi_shared(...); // becomes assoc. with access region
}

// unrelated explicit-handle operation not assoc. with access region
h2 = gasnet_get_nb_shared(...);
gasnet_wait_syncnb(h2);

// end the access region and get the handle
handle = gasnet_end_nbi_accessregion();

.... // other code, which may include unrelated implicit-handle
     // operations+syncs, or other regions, etc

// wait for all the operations assoc. with access region to complete
gasnet_wait_syncnb(handle);
```

3.4 Register-memory operations

Register-memory operations allow client code to avoid forcing communicated data to pass through the local memory system. Some interconnects may be able to take advantage of this capability and launch remote puts directly from registers or recieve remote gets directly into registers.

3.4.1 Value Put

3.4.1.1 gasnet_put_val, gasnet_put_nb_val, gasnet_put_nbi_val

`void` **gasnet_put_val** (`gasnet_node_t` *node*, `void *`*dest*,
`gasnet_register_value_t` *value*, `size_t` *nbytes*);
`gasnet_handle_t` **gasnet_put_nb_val** (`gasnet_node_t` *node*, `void *`*dest*,
`gasnet_register_value_t` *value*, `size_t` *nbytes*);
`void` **gasnet_put_nbi_val** (`gasnet_node_t` *node*, `void *`*dest*,
`gasnet_register_value_t` *value*, `size_t` *nbytes*);

Register-to-remote-memory put - these functions take the value to be put as input parameter to avoid forcing outgoing values to local memory in client code. Otherwise, the behavior is identical to the memory-to-memory versions of put above. Requires: `nbytes > 0 && nbytes <= SIZEOF_GASNET_REGISTER_VALUE_T`. The value written to the target address is a direct byte copy of the 8**nbytes* low-order bits of value, written with the endianness appropriate for an *nbytes* integral value on the current architecture. The non-blocking forms of value put must be synchronized using the explicit or implicit synchronization functions defined above, as appropriate

3.4.2 Blocking Value Get

3.4.2.1 gasnet_get_val

`gasnet_register_value_t` **gasnet_get_val** (`gasnet_node_t` *node*, `void *`*src*, `size_t` *nbytes*);

This function returns the fetched value to avoid forcing incoming values through local memory (on architectures which pass the return value in a register). Otherwise, the behavior is identical to the memory-to-memory blocking get. Requires: `nbytes > 0 && nbytes <= SIZEOF_GASNET_REGISTER_VALUE_T`. The value returned is the one obtained by reading the *nbytes* bytes starting at the source address with the endianness appropriate for an *nbytes* integral value on the current architecture and setting the high-order bits (if any) to zero (i.e. no sign-extension)

3.4.3 Non-Blocking Value Get (explicit-handle)

This operates similarly to the blocking form of value get, but is split-phase. Non-blocking value gets are synchronized independently of all other operations in GASNet.

`typedef ??? gasnet_valget_handle_t;`

3.4.3.1 gasnet_get_nb_val, gasnet_wait_syncnb_valget

`gasnet_valget_handle_t` **gasnet_get_nb_val** (`gasnet_node_t` *node*, `void *`*src*, `size_t` *nbytes*);
`gasnet_register_value_t` **gasnet_wait_syncnb_valget** (`gasnet_valget_handle_t` *handle*);

`gasnet_get_nb_val` initiates a non-blocking value get and returns an explicit handle which **must** be synchronized using `gasnet_wait_syncnb_valget()` `gasnet_wait_syncnb_valget()` synchronizes an outstanding `get_nb_val` operation and returns the retrieved value as described for the blocking version. Note that `gasnet_valget_handle_t` and `gasnet_handle_t` are completely different datatypes and may not be intermixed (i.e. `gasnet_valget_handle_t`'s cannot be used with other explicit synchronization functions, and `gasnet_handle_t`'s cannot be passed to `gasnet_wait_syncnb_valget()`. The `gasnet_valget_handle_t` type is completely opaque (with no special "invalid" value), although implementors are recommended to make `sizeof(gasnet_valget_handle_t) <= sizeof(gasnet_register_value_t)` to facilitate register reuse. There is no try variant of value get synchronization, and no implicit-handle variant.

3.5 Barriers

The following functions can be used to execute a parallel split-phase barrier with the given barrier identifier across all nodes in the job. Note that the barrier wait/notify functions should only be called once (i.e. by one representative thread) on each node per barrier phase. The client must synchronize its own accesses to the barrier functions and ensure that only one thread is ever inside a GASNet barrier function at a time (esp. `gasnet_barrier_try()`).

```
#define GASNET_BARRIERFLAG_ANONYMOUS ???
#define GASNET_BARRIERFLAG_MISMATCH ???
```

3.5.0.1 gasnet_barrier_notify

`void` **gasnet_barrier_notify** (`int` *id*, `int` *flags*)

Execute the notification for a split-phase barrier, with a barrier value *id*. This is a non-blocking operation that completes immediately after noting the barrier value. No synchronization is performed on outstanding non-blocking memory operations.

Generates a fatal error if this is the second call to `gasnet_barrier_notify()` on this node since the last call to `gasnet_barrier_wait()` or the beginning of the program.

If *flags* == 0 then this is a "named" barrier notify that carries the given *id* value. If *flags* == `GASNET_BARRIERFLAG_ANONYMOUS`, then *id* is ignored and the barrier is anonymous - it has no specific value. If *flags* == `GASNET_BARRIERFLAG_MISMATCH`, then the subsequent `gasnet_barrier_wait()` call on every node will return `GASNET_ERR_BARRIER_MISMATCH` (i.e. allows the client to force a global mismatch error when a mismatch was detected locally).

3.5.0.2 gasnet_barrier_wait

`int` **gasnet_barrier_wait** (`int` *id*, `int` *flags*)

Execute the wait for a split-phase barrier, with a barrier value. This is a blocking operation that returns only after all remote nodes have called `gasnet_barrier_notify()`. No synchronization is performed on outstanding non-blocking memory operations .

Generates a fatal error if there were no preceding calls to `gasnet_barrier_notify()` on this node, or if this is the second call to `gasnet_barrier_wait()` (or successful call to `gasnet_barrier_try()`) since the last call to `gasnet_barrier_notify()` on this node. On a `GASNET_PAR` or `GASNET_PARSYNC` configuration, the thread calling `gasnet_barrier_notify()` is permitted to differ from the thread which calls the paired `gasnet_barrier_wait()`, but the ordering between the calls must still be maintained.

Returns `GASNET_ERR_BARRIER_MISMATCH` if *flags* is not equal to the *flags* value passed to the preceding `gasnet_barrier_notify()` call made by this node. Returns `GASNET_ERR_BARRIER_MISMATCH` if the *flags* value passed to `gasnet_barrier_notify()` on this or any other node was `GASNET_BARRIERFLAG_MISMATCH`. Returns `GASNET_ERR_BARRIER_MISMATCH` if *flags*==0 and the supplied *id* value doesn't match the *id* value provided in the preceding `gasnet_barrier_notify()` call made by this node. Returns `GASNET_ERR_BARRIER_MISMATCH` if any two nodes passed non-anonymous barrier values which didn't match during the `gasnet_barrier_notify()` calls which began this barrier phase. Otherwise, returns `GASNET_OK` to indicate that all nodes have called a matching `gasnet_barrier_notify()` and the barrier phase is complete.

3.5.0.3 gasnet_barrier_try

`int` **gasnet_barrier_try** (`int` *id*, `int` *flags*)

`gasnet_barrier_try()` functions similarly to `gasnet_wait()`, except that it always returns immediately. If the barrier has been notified by all nodes, the call behaves as a call to `gasnet_barrier_wait()` with the same barrier *id* and *flags*, and returns `GASNET_OK` (or `GASNET_ERR_BARRIER_MISMATCH` in the case a mismatch is detected). If the barrier has not yet been notified by some node, the call is a no-op and returns the value `GASNET_ERR_NOT_READY`.

Generates a fatal error if there were no preceding calls to `gasnet_barrier_notify()` on this node, or if this is the second call to `gasnet_barrier_wait()` (or successful call to `gasnet_barrier_try()`) since the last call to `gasnet_barrier_notify()` on this node.

3.6 Threading support

3.6.1 Thread-identification optimization:

When compiled in the `GASNET_PAR` or `GASNET_PARSYNC` configurations, GASNet is capable of handling multiple client threads. It is likely that GASNet implementations will need to distinguish these threads, specifically they may need to store some metadata associated with each client thread. Unfortunately, the overhead of discovering the identity of a particular client thread making a GASNet call (hereafter termed "thread discovery") can have a non-trivial overhead on some threading systems (e.g. the cost of calling `pthread_self()` or `pthread_getspecific()`). Many of the simpler GASNet functions could have their performance dominated by this cost if they need to perform thread discovery on every call.

The following macros provide a way for the client to amortize the cost of thread discovery over many GASNet calls made by the same thread. This is an optimization which is *totally* optional - clients need not make any of the calls below to have a working system, although GASNet performance is likely to suffer without it in a `GASNET_PAR` or `GASNET_PARSYNC` configuration.

```
typedef void *gasnet_threadinfo_t;
```

`gasnet_threadinfo_t` is an opaque pointer representing the internal GASNet metadata associated with a particular client thread.

3.6.1.1 GASNET_GET_THREADINFO

```
#define GASNET_GET_THREADINFO() ???
```

Returns a value of type `gasnet_threadinfo_t` which represents the GASNet internal metadata associated with the current client thread. This `gasnet_threadinfo_t` value can be passed into or out of functions and may be posted for GASNet's use with `GASNET_POST_THREADINFO()`. May be called from anywhere in the client program, at any time after GASNet initialization. It is erroneous to hand-off this `gasnet_threadinfo_t` value to a different client thread.

3.6.1.2 GASNET_POST_THREADINFO

```
#define GASNET_POST_THREADINFO(info) ???
```

This macro may *optionally* be placed at the top of functions which make calls to GASNet. It has no runtime semantics, but it may provide a performance boost on some implementations (especially in functions which make multiple calls to the extended API - e.g. it provides the implementation with a place for minimal per-function initialization or temporary storage that may be helpful in amortizing implementation-specific overheads). When used, it must appear only at the very beginning of a function or block (before any declarations or calls to the API in that function). It may not appear as a global declaration. The info argument must be a `gasnet_threadinfo_t` value acquired from a previous call to `GASNET_GET_THREADINFO()` on this thread.

3.6.1.3 GASNET_BEGIN_FUNCTION

```
#define GASNET_BEGIN_FUNCTION() ???
```

A convenience macro that may *optionally* be placed at the top of functions which repeatedly make GASNet calls, to amortize the overhead of thread discovery on some implementations.

It has behavior equivalent to `GASNET_POST_THREADINFO(GASNET_GET_THREADINFO())`, however some implementations may choose to lazily postpone performing thread discovery until the first place where it is actually needed.

3.6.2 Thread management

3.6.2.1 gasnet_set_waitmode

`int` **gasnet_set_waitmode** (`int` *wait_mode*)

Optional call which gives the GASNet implementation a hint about how aggressively threads within blocking GASNet calls should contend for CPU resources. *wait_mode* must be one of the following recognized values:

`GASNET_WAIT_SPIN`
: contend aggressively for CPU resources while waiting (spin)

`GASNET_WAIT_BLOCK`
: yield CPU resources immediately while waiting (block)

`GASNET_WAIT_SPINBLOCK`
: spin for an implementation-dependent period, then block

Wait mode is a per-node hint which is permitted to differ across GASNet nodes.

Returns `GASNET_OK` on success.

Appendix A Notes

A.1 Open Issues in the GASNet Specification

- Add support for strided & scatter/gather accesses (spec underway - will be included in GASNet spec v2.0)
- Add collective ops - reductions, scans, etc. (spec underway - will be included in GASNet spec v2.0)
- Is it worthwhile to add a blocking barrier call? May want to tune this differently from split-phase barrier on some networks, but would need to define the semantics of how blocking and non-blocking barriers can be used together.

A.2 Core API Active Messaging Functions - differences from Active Messages 2.0

The GASNet core API was originally based on Active Messages 2.0 (as described in *A. Mainwaring and D. Culler in "Active Message Applications Programming Interface and Communication Subsystem Organization"*), however we've removed some of the generality which is not required (and can lead to performance degradation and more implementation effort), and stripped it down to the bare essentials required for active messages in a purely SPMD environment. The final spec more closely resembles the "*Generic Active Message Interface Specification v.1.1*", *by D.Culler et al.*, however we describe the differences from AM2.0 for readers familiar with that specification (and because we envision a number of the GASNet core implementations being simply a thin wrapper over the existing AM2.0 implementations on a number of platforms).

Here are a summary of the changes (informal style.. this is not really part of the spec):

- the functions are renamed to match the GASNet conventions
- there are no bundles and only one (implicit) endpoint. This necessitates the following changes:
  - All AM2 functions which took an endpoint or bundle argument have that argument removed
  - The following functions no longer exist: `AM_Init`, `AM_Terminate`, `AM_AllocateBundle`, `AM_AllocateEndpoint`, `AM_FreeEndpoint`, `AM_FreeBundle`, `AM_MoveEndpoint`, `AM_GetXferM`, `AM_GetDestEndpoint`
- all handler registration is performed during `gasnet_attach()`, and the maximum number of handlers is fixed at 256 (including handler 0, the error handler)
  - The following functions no longer exist: `AM_SetHandler` and `AM_SetHandlerAny`, `AM_GetNumHandlers`, `AM_SetNumberHandlers`, `AM_MaxNumHandlers`
- Segment registration is handled by `gasnet_attach()` (using a `uintptr_t` to allow entire VA space)
  - The following functions no longer exist: `AM_SetSeg` and `AM_MaxSegLength` (still have `AM_GetSeg`)
  - implementations must support an endpoint segment length that spans the entire virtual address space, though the performance may change for larger segment sizes (if `gasnet_attach` requests a size larger than what underlying `AM_SetSeg` can provide, then we turn off large AM Xfers and emulate gasnet_Xfer using medium messages)
  - the `dest_offset` argument to the Xfer functions is changed to a `void *` address
- there are no tags or endpoint names visible to the user - such details are all handled internally by the job startup mechanism, which sets up a SPMD-style mapping table (all the nodes, including the current node, in ascending order by rank).
  - Therefore, the following functions no longer exist: `AM_Map`, `AM_MapAny`, `AM_Unmap`, `AM_SetTag`, `AM_GetTag`, `AM_GetTranslationName`, `AM_GetTranslationTag`, `AM_GetTranslationInUse`, `AM_MaxNumTranslations`, `AM_GetNumTranslations`, `AM_SetNumTranslations`, `AM_GetMsgTag`
  - the `en_t *` argument to `AM_GetSourceEndpoint` is now an `gasnet_node_t *` and returns the node rank of the sender (the now-opaque token could be implemented as the integer node index itself, although we allow implementations to still use it as a ptr to metadata if required)
- `AM_RequestXferAsyncM` has more useful semantics (may block)

- `AM_SetExpectedResources` no longer exists
- all implementations must support the `AM_PAR` (multi-threaded) access mode (`GASNET_PAR` configuration)
- we handle 64-bit implementations - require small size to be 16 32-bit args (ensure 8 `(void*)`'s can be sent)
  cons: handler code needs to be rewritten for 64-bit platforms to perform packing/unpacking
- Blocking polling operation is simplified in the following ways:
  - `AM_GetEventMask` and `AM_SetEventMask` no longer exist
  - `AM_WaitSema` is replaced with `GASNET_BLOCKUNTIL()`
- Maybe deprecate `ReplyXfer` in favor of `GetXfer`
  - some implementations have trouble with large `ReplyXfer`'s (with software flow control & reliability)
  - better yet, just separate `AM_MaxLong` into `AM_MaxLongRequest`, and `AM_MaxLongReply`
  - AM2.0 `GetXfer` doesn't add any expressiveness - really want a way to get from remote segment into arbitrary local memory address
- All Xfer functions specify the destination using a virtual memory address (which must fall within the registered segment) rather than a segment offset.
- request handlers are permitted to omit a reply call if no reply handler is needed (and some implementations may optimize this case)

A.3 Active Message Categories - Alternate formulation of AM (not part of spec)

Newcomers to Active Messages and GASNet occasionally express confusion over the concepts of Short, Medium and Long AM's. Despite the somewhat misleading naming convention, the three categories of messages may actually bear only a loose correllation to the actual message/data sizes. The important distinctions are semantic, and sufficiently minor that one might imagine replacing the three categories with a single, more general type of AM that provides the functionality of each GASNet AM category as a special case.

Specifically, a GASNet Short AM can be seen as a special case of a Medium or Long AM where the payload has length zero. Furthermore, the only important semantic distinction between Medium and Long AM's are that Medium AM's provide the payload to the handler in a temporary network buffer, whereas Long AM's write the payload (often using RDMA) to a sender-specified location in the user memory segment of the target node before running the handler (each semantic is useful for different usage scenarios).

Hence, some users may find it helpful to consider building "unified" AM request/reply functions such as suggested below:

```
/* unified request function */
int unified_AMRequestM(
       gasnet_node_t dest, gasnet_handler_t handler,
       void *buf, size_t buf_len,
       void *dest_addr,
       int32 arg0, int32 arg1, ...) {
  if (buf == NULL)
    return gasnet_AMRequestShortM(dest,handler,arg0,arg1,...);
  else if (dest_addr == NULL)
    return gasnet_AMRequestMediumM(dest,handler,buf,buf_len,arg0,arg1,...);
  else
    return gasnet_AMRequestLongM(dest,handler,buf,buf_len,dest_addr,arg0,arg1,...);
}
```

- M is the number of arguments, which must be <= `gasnet_AMMaxArgs()`
- `dest_addr == NULL` requires `buf_len` <= `gasnet_AMMaxMedium()`, and the payload is delivered in a temporary buffer
- `dest_addr != NULL` requires `buf_len` <= `gasnet_AMMaxLongRequest()` (or `gasnet_AMMaxLongReply()` for replies), and the payload is written into the target node segment at `dest_addr`

- Handler prototypes remain the same as under GASNet:
  `buf == NULL`:

```
void handler_nopayload(gasnet_token_t token,
                       gasnet_handlerarg_t arg0, gasnet_handlerarg_t arg1...);
```

  `buf != NULL`:

```
void handler_withpayload(gasnet_token_t token,
                         void *buf, size_t buf_len,
                         gasnet_handlerarg_t arg0, gasnet_handlerarg_t arg1...);
```

Concept Index

Function, Macro and Type Index

G