GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages

CS258 Parallel Computer Architecture Project, Spring 2002

Dan Bonachea (bonachea@cs) and Jaein Jeong (jaein@cs)

Abstract

Global Address Space (GAS) languages are an important class of parallel programming languages that provide a shared-memory abstraction to the programmer on arbitrary hardware. Efficient implementation of GAS languages (such as UPC) is critically dependent on the use of low-latency, low-overhead, high-bandwidth communication systems – unfortunately most low-level high-performance communication interfaces are highly vendor and machine specific, which is a serious impediment to compiler portability. We present the GASNet communication API, which provides an expressive and portable interface for high-performance one-sided communication that can be used as a compilation target for GAS languages. We have completed a prototype implementation of the GASNet API on MPI, and present initial micro-benchmark performance results for the prototype – we find that MPI itself is not a very suitable target for implementing GAS languages, but that the performance overheads imposed by the use of the GASNet API are minimal.

1 Introduction

Modern parallel architectures can be roughly divided into two camps based on the programming interface exposed by the hardware: shared memory systems where parallel threads of control all share a single logical memory space (and communication is achieved through simple loads and stores), and distributed memory systems where some (but not necessarily all) threads of control have disjoint memory spaces and communicate through explicit communication operations (e.g. message passing). Experience has shown that the shared memory model is often easier to program with and reason about, however it presents serious scalability challenges to hardware designers and (with a few notable exceptions) distributed memory machines currently dominate the high-end supercomputing market (i.e. systems with 100's to 1000's of compute processors).

The global-address space model is a hybrid that seeks to combine the advantages of both models. It offers the programmability advantages of a globally shared address space, but is carefully designed to allow efficient implementation on distributed-memory message-passing architectures. UPC, Titanium and Co-array Fortran are examples of modern programming languages that provide a global-address space memory model. GAS languages typically make the distinction between local and remote memory references explicitly visible to encourage programmers to consider the locality properties of their program.

The next section describes the UPC programming language. Section 2 presents the architecture of our UPC compiler and the GASNet communication system. Section 3 discusses the novel support for handler atomicity provided by the GASNet core API. Section 4 provides initial performance results for the prototype GASNet implementation. Section 5 discusses related work, and we conclude and summarize future work in section 6.

1.1 UPC

UPC [11] is a parallel SPMD superset of the C programming language. It provides a shared memory abstraction to the programmer, regardless of the memory model provided by the underlying hardware. Experience has shown that in order to design high-performance parallel algorithms for high-end supercomputers (especially those with physically distributed memory) it is important to consider the locality properties of the major data structures and data accesses within the inner loops - even in the presence of a shared-memory abstraction. Towards this goal, the UPC memory space is logically divided into a "private" area associated with each thread (and accessible only to that thread), and a shared memory area (accessible from any thread), as shown in Figure 1. In addition, all shared memory objects have "affinity" to a specific thread that will have the fastest possible access to that object (in a distributed memory system, this means the object resides in the local memory of the associated compute processor).

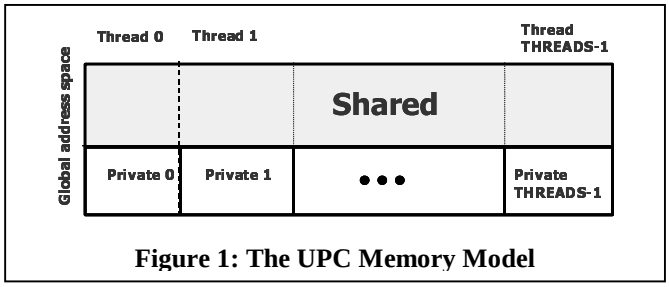


Figure 1: The UPC Memory Model

UPC has language support for parallel data distribution - arrays in shared memory may have their affinity distributed across one or more threads (as specified by the programmer) so a specific subset of the elements has affinity to a given thread. A new parallel loop construct (upc_forall) allows programmers to easily specify a collective computation where each thread operates on the data with affinity to that thread. Finally, there is language support for relaxed memory consistency models to enable aggressive use of non-blocking remote memory operations that help tolerate network latencies.

UPC is a relatively new programming language (the first mature specification was standardized in Feb, 2001), and compiler development efforts are underway at a number of major corporations and institutions [26]. With one notable exception ([21]), all efforts to-date focus on implementing a monolithic compiler that translates UPC programs directly to machine code (generally by extending an existing C compiler). For example, the first UPC compiler [6] modified the code-generation phase of the GNU gcc compiler to implement UPC memory references on the Cray T3E using the platform-specific E-registers. The monolithic compiler approach has the advantage of preserving information gleaned from high-level language constructs down to the assembly code level, but has the serious disadvantage that the compiler must be re-implemented from scratch for each platform and network architecture. This negatively affects widespread acceptance of the language and possibly hinders the portability of applications.

2 System Architecture

The UPC project at NERSC [15] (a joint effort between LBL and UC Berkeley, funded in part by the DOE pmodels grant) is seeking to develop a fully-portable, high-performance UPC compiler that will run on a wide variety of shared-memory and distributed-memory platforms using different network interconnects, including large-scale multiprocessors, PC clusters, and clusters of shared memory multiprocessors. One of the main goals of the project is to experiment with parallel compiler optimization techniques, without being tied to a particular system architecture or network. Portability is achieved by translating UPC programs to an intermediate representation in C, which can then be compiled using the system's ANSI C compiler and linked to a standardized runtime system and communication system tailored to the specific platform.

Figure 2 shows the high-level system diagram for a UPC application compiled using the NERSC UPC compiler. The generated C code will run on top of the UPC runtime system, which will provide platform independence and implement language-specific features such as shared memory allocation and shared pointer manipulation. The runtime system will implement remote operations by calling the GASNet communication interface, which provides hardware-independent lightweight networking primitives.

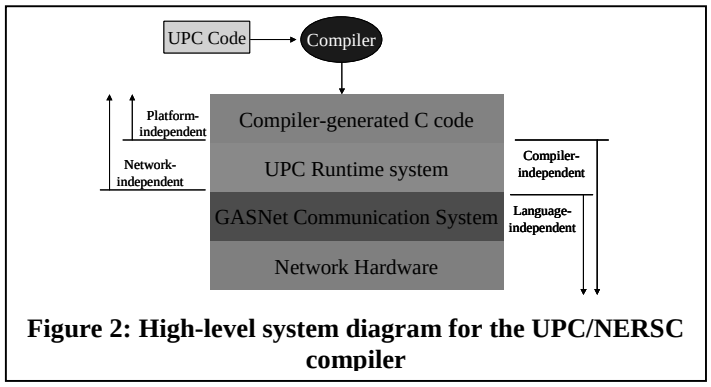


Figure 2: High-level system diagram for the UPC/NERSC compiler

2.1 GASNet Design

The primary goal of GASNet is to provide a high-performance, network-independent communication interface that can be used as a compilation target for any global address-space programming language. The performance goal is an obvious one – programmers generally write parallel applications because they care about performance, and the overall performance of applications written in GAS languages can be very sensitive to network performance characteristics (especially small message latency and overhead). Unfortunately, most low-level high-performance communication interfaces are highly vendor and machine specific - which is a serious impediment to compiler portability. GASNet abstracts away the network-specific details to provide a portable, yet still high-performance interface - a GAS language compiler using GASNet can be retargeted to a new network system by simply implementing the GASNet API on the new network.

The design of GASNet is partitioned into two layers to maximize porting ease without sacrificing performance: the lower level is a narrow but very general interface called the GASNet core API – the design is based heavily on Active Messages [17], and is implemented directly on top of each individual network architecture. The upper level is a wider and more expressive interface called the GASNet extended API, which provides high-level operations such as remote memory access and various collective operations. We've written a network-independent reference implementation of the extended API purely in terms of the core API, which allows GASNet (and the GAS compiler) to quickly and easily be

ported to a new network architecture by re-implementing only the minimal core API. GASNet is structured such that implementers can choose to additionally bypass certain functions in the reference implementation of the extended API and implement them directly on the hardware to improve performance of specific operations when hardware support is available (e.g. special network support for puts/gets or hardware-assisted broadcast).

The complete GASNet specification we created is included as Appendix A, however the highlights will be presented here.

2.2 GASNet Core API

The GASNet core API is a narrow interface based on the Active Messages (AM) paradigm [28], which is general enough to implement everything in the extended API. The AM 2.0 specification [17] was a starting point for the design – however, we've stripped out many of the extraneous complexities (e.g. multiple endpoint support, explicit translation management) which are irrelevant in the context of a SPMD global-address space communication system. We've also extended the interface somewhat by adding support for 64-bit architectures and mechanisms for explicit handler atomicity control (see section 3). Finally, the core API includes direct support for SPMD job bootstrapping and layout queries, a feature often omitted or overlooked in low-level communication interfaces that generally leads to lots platform-specific or even site-specific bootstrapping code in the client.

Active Messages is basically a low-level super-lightweight RPC mechanism – it provides unordered, reliable delivery of matched request/reply messages that are serviced by user-provided lightweight handlers, which run quickly and to completion to integrate communicated data into the ongoing computation. All active messages may carry a small number of integer arguments to the user-provided handler. Additionally, "medium"-sized active messages carry a payload of opaque data made available to the handler in a temporary buffer, and "long"-sized active messages carry a DMA transfer of data that is transferred to a location specified by the sender before the handler runs. AM implementations are available for a number of network architectures (Via, Myrinet, MPI, LAPI, UDP, etc.), and the AM paradigm has strongly influenced the design of a number of other network interfaces (e.g. LAPI, GM).

Native AM implementations on high-performance networks are usually implemented as a purely user-level communication system to maximize latency performance, and we expect the same will be true for GASNet core implementations. A variety of message receipt and servicing paradigms are provided by modern high-performance network hardware, however AM 2.0 requires handlers to run in a polling-based manner, that is, handlers for pending messages run synchronously on the compute thread during calls to AM_Poll() or other AM message send operations. Our GASNet core API relaxes this requirement and allows core implementors to select the message handling technique most appropriate for the given network. Examples of alternative message reception techniques include: using hardware interrupts to interrupt the computation thread (especially useful for systems that support fast user-level network interrupts, such as the J-machine [10]), combinations of interrupts and polling (such as the Polling Watchdog [18]), keeping a private, asynchronous communication thread which performs all message reception and handler execution (similar to LAPI's completion thread [12]), and more exotic alternatives such as dedicated NIC processors capable of running arbitrary handler code (similar to the FLASH multiprocessor [14]). One consequence of this flexibility is that the core must provide additional mechanisms to enable the implementation of AM handlers that atomically update critical data structures – this issue is discussed further in section 3.

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 AM 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.). Experience has shown that AM is general enough to express most or all of the communication patterns desirable in implementing GAS languages – therefore in addition to the portability benefits, the availability of the core API also provides a nice extensibility mechanism to accommodate unforeseen future communication needs.

2.3 GASNet Extended API

The GASNet extended API is a richly expressive and flexible interface that provides high-level one-sided remote memory get/put operations and collective operations (basically any primitive that would be useful for implementing a GAS language that we could imagine being implemented using hardware support on some NICs).

The remote memory operations in the GASNet extended API were designed to support a wide variety of usage scenarios to facilitate experimentation with parallel compiler optimizations. They include simple blocking gets/puts, and two flavors of non-blocking data transfers. Explicit-handle non-blocking operations ("nb" suffix) return a handle representing the operation in-flight, and must later be completed by passing this handle to an explicit-handle synchronization function (there are functions for synchronizing on a particular handle, or a specific list of handles). Implicit-handle non-blocking operations ("nbi" suffix) do not return a handle, and are completed using generic synchronizations functions that synchronize on the completion of all the current thread's outstanding implicit-handle puts, gets, or both. Access region synchronization provides a mechanism for associating a single explicit handle with all the implicit-handle operations initiated during a given time interval. Every synchronization function supports polling or blocking for completion. All of the data transfer functions provide memory-to-memory transfers, and many also provide register-to-memory or memory-to-register transfers, to avoid forcing data to pass through local memory on architectures that support remote memory gets/puts directly to/from registers (such as the Cray T3E). Remote memory addresses are expressed using a tuple of (node rank, virtual address), and all the operations in the extended and core API support loopback[1].

The only collective operation currently provided in the extended API is a named split-phase barrier, although we plan to extend this list to include other operations such as broadcasts, reductions, scan etc. in the near future (we are waiting on a new version of the UPC specification which will incorporate these features at the language level).

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. 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 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).

[1] Although in many cases the loopback cutoff will likely occur at a higher level, in the GAS language runtime system.

3 Handler Atomicity

Traditional AM 2.0 implementations are purely polling-based, meaning that all handlers run synchronously on the main thread of computation during calls to AM_Poll() or the message sending functions. Handlers must run to completion and are never permitted to make blocking calls (such as lock acquire) to prevent deadlocking the system. Therefore, the only way to achieve atomicity within AM 2.0 handlers (for example, to atomically update a shared data structure) is to only make AM calls from a single application thread and only while holding a lock that also protects all the data structures that may be accessed by handlers – the synchronous nature of handler dispatch then ensures all handlers will run atomically with respect to each other and the main threads of computation that may be modifying the data structures in question. This approach works but greatly limits the level of concurrency possible in the system (especially in the presence of multiple processors) because it over-synchronizes the handlers (enforces atomicity even when it is not required), and creates a bottleneck between application threads on the single lock that protects all handler-accessible data structures and the network.

GASNet relaxes the restrictions on handler dispatch to allow interrupt-based handlers and other more general message reception techniques. In GASNet, core handlers may run asynchronously with respect to all application threads, and they may run concurrently with each other on separate threads – the only restriction is that handlers are not interruptible (that is, any given thread will be running at most a single handler at a time and will never be interrupted to run another handler). The fully asynchronous nature of handlers implies that even the non-optimal trick described above is insufficient for achieving handler atomicity under GASNet. Therefore, GASNet core provides two (related) mechanisms to make handlers safe and provide atomicity when required: No-Interrupt Sections ensure signal safety of handler code for GASNet implementations using interrupt-based mechanisms, and a Handler-Safe Locks allow atomic updates from handlers to data structures shared with the client threads and other handlers.

3.1 The problem

Figure 3 illustrates one of the basic problems of why it can be unsafe for handlers to block for lock acquisition. With interrupt-based handler dispatch, there's nothing to prevent such a handler from running in the context of a

thread that was interrupted while holding the same lock. This situation would cause system deadlock because the handler will block forever waiting for the outer critical section to release the lock, but this cannot happen because the handler interrupt is preventing that critical section from making progress. A similar problem arises under polling-based handler dispatch if the application thread attempts to poll or send messages (which implies polling) from within a critical section – under this scenario, the handler runs synchronously with respect to the application thread, but the potential for deadlock remains.

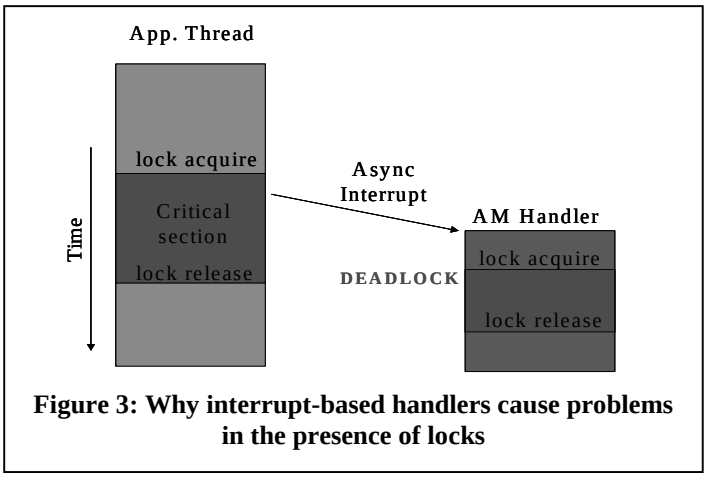


Figure 3: Why interrupt-based handlers cause problems in the presence of locks

A different, but closely related problem occurs with library calls in the presence of interrupt-based handler dispatch, where handlers may interrupt a call to a non-reentrant library function from the main application thread. If the handler then attempts to call the same library function, the internal library data structures are likely to become corrupted. This problem is analogous to the problem of calling library functions from within UNIX signal handlers (in fact, interrupt-based GASNet cores are likely to use signals to interrupt the main computation and run handlers). Note that even most "thread-safe" libraries are unlikely to also be reentrant, and will break or deadlock if called from a handler interrupt 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(), which may be used by handlers to perform more sophisticated, multi-message data transfer operations (such as scatter-gather).

3.2 No-Interrupt Sections

To overcome the problem of interrupts during non-reentrant library calls, and permit our handlers to be more useful, the GASNet core allows the client to temporarily disable interrupt-based handler execution on a specific thread. All calls to non-reentrant functions that could possibly access state shared by functions also called from handlers *must* be called within a GASNet "No-Interrupt Section":

```
gasnet_hold_interrupts();
    some_nonreentrant_library_call(…);
gasnet_resume_interrupts();
```

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 private GASNet threads. 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 from inside a call to a non-signal-safe library. This is a change from other systems (such as von Eicken's original AM-1 implementation [28] and Alewife's two-case delivery system [16]) that completely disable all hardware network interrupts to achieve atomicity – GASNet relaxes this restriction to support more concurrency on SMP nodes where we may safely continue taking interrupts and running handlers on other threads.

The GASNet core implementation guarantees the client that handlers will not run asynchronously within a No-Interrupt Section, but it is the client's responsibility to never call GASNet functions within the section that could cause handlers to execute synchronously[2]. No-interrupt sections also have the potential to reduce node responsiveness to the network, so there is a set of conventions restricting client code behavior within a No-Interrupt Section:

- Code in a No-Interrupt Section must not call any GASNet functions that may send requests or synchronously run handlers
- hold/resume should not be called from within a handler context - handlers are run within an implicit No-Interrupt Section
- 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 (could cause deadlock)
- 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 could cause some GASNet

[2] This is not actually a problem for reentrancy safety, but it is an issue for Handler-Safe Locks, described in the next section.

implementations employing timeout-based mechanisms to fail (e.g. remote nodes may decide this node is dead and abort the job).

- No-Interrupt Sections may not be nested (for implementation convenience and efficiency)

GASNet core implementations that never use interrupts to run handlers can implement hold/resume as no-ops – this is good because the application threads may be calling hold/resume operations with some frequency and therefore they should be optimized for low overhead. Cores that use interrupt-based dispatch only need to ensure that handlers never run on a thread that is in the interrupts-disabled state – one naïve way to implement this is to disable all hardware interrupts while any thread has interrupts disabled, but this requires some coordination between threads and there may be non-trivial overheads associated with disabling/enabling interrupts on the NIC (for example, each transition may involve a kernel call and/or a system bus transaction).

A more clever way to implement interrupt hold/resume for cores with interrupt-based handler dispatch is to use a multi-threaded extension of von Eicken's interrupt handshake [29] – basically, the core keeps two bits of state in memory for each application thread: an interruptsDisabled bit and a messageArrived bit. The hold operation merely clears the messageArrived bit and sets the interruptsDisabled bit (can be done with a single memory write to a location which is likely to be cached). The interrupt handler always checks the interruptsDisabled bit for the current thread before running handlers, and if it happens to be set it merely sets the messageArrived bit (to indicate a missed interrupt) and exits. The resume operation clears the interruptsDisabled bit, and if the messageArrived bit indicates an interrupt was missed it synchronously polls the network to service any number of waiting messages. This allows us to temporarily defer incoming messages during the No-Interrupt Section.

Because the No-Interrupt Section is meant to be very short in real-time, the expected common case is that no message interrupts will arrive during that interval - however even if they do, the messages won't be deferred for very long (i.e. no longer than until the end of the No-Interrupt Section). On many implementations (where the NIC is a dedicated application resource) it may be perfectly acceptable to leave the messages in the incoming hardware FIFO queue during this interval[3]. Also note that nothing prevents *other* threads in a multi-threaded client or core implementation from synchronously servicing the messages that caused the interrupt – this is good because other threads which are spin-waiting for a network result won't have their observed latency affected by the fact that a different thread may be inside a No-Interrupt Section.

3.3 Handler-Safe Locks

In order to support handlers that need to atomically update data structures accessed by the main-line client code and other handlers, the GASNet core 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 that may be safely acquired from within a handler context. All lock-protected data structures in the client that need to be accessed by handlers should be protected using an HSL (i.e. instead of a standard system lock). The interface to HSL's is similar to the POSIX mutex interface – HSL's can be allocated statically or dynamically, and there are lock() and unlock() functions (although no trylock()).

Similar to No-Interrupt Sections, there is a set of coding conventions that restrict the usage of HSL's, which allows them to be deadlock-free when acquired from handlers. The basic idea is that we enforce a No-Interrupt Section over any thread holding at least one HSL (to prevent the execution of synchronous or asynchronous handlers that may try to acquire the same HSL, leading to deadlock as illustrated in figure 3), and we require that HSL's always be held for a "bounded" amount of time, so that handlers which are blocking to acquire an HSL currently held by a different thread are guaranteed to only block for bounded amount of time:

- 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 above). hold/resume must not be explicitly called while holding an HSL
- Any handler which locks one or more HSL's *must* unlock them all before exiting or sending a reply

The next two restrictions are not necessary for safety, but are added to help maximize efficiency of the implementation:

[3] Systems where the network must be continually serviced to ensure system progress (e.g. where the same network carries cache coherency messages) may require temporarily buffering user messages during this interval.

- HSL's may *not* be locked recursively, although it *is* permitted for a thread to acquire more than one HSL[4].
- 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)

Handler-Safe Locks safely provide explicit atomicity control to AM handlers, even in the presence of interrupt-based and/or multi-threaded concurrent handler servicing policies. Most importantly, they don't over-synchronize AM handlers and only provide the necessary level of atomicity (which is good because most critical path AM handler probably do *not* need atomicity). Finally, the opaque HSL's can be implemented very efficiently using a simple system lock and a small amount of additional state (the exact state information required is based on the specific policies of the given core implementation).

The safety of HSL's really comes from the set of restrictions governing their usage. We only expect experienced language implementors to be writing AM handlers and using HSL's, so it is reasonable to impose such usage conventions on HSL's. However in the interests of assisting these implementors, our prototype GASNet core implementation includes a debugging mode that adds extra runtime checks to detect if any of the usage conventions on No-Interrupt Sections or HSL's are violated, thereby eagerly enforcing the conditions which provide deadlock-freedom, rather than relying on testing to actually deadlock the system and then debug what violation led to the problem.

4 Results

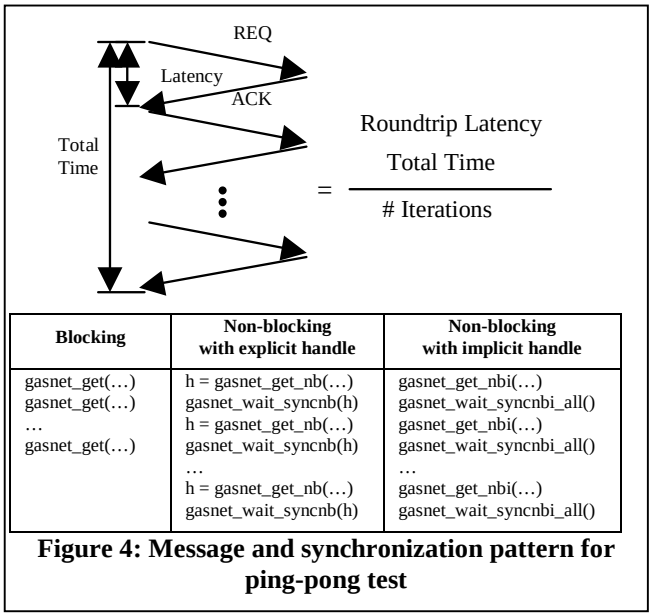
To evaluate the effectiveness of GASNet, we implemented a prototype GASNet core as a wrapper around AMMPI [1], a portable AM 2.0 implementation over MPI. The prototype uses our unmodified reference implementation of the extended API. The portability of MPI (and AMMPI) means this entire GASNet implementation is trivially portable, and we were able to run performance experiments on an IBM SP system and a Linux-x86/Myrinet cluster (Millennium) merely by changing the Makefile. Because AMMPI is simply a translation layer between AM and MPI and not a native implementation of AM, we don't expect stellar absolute performance results – however, we can still examine the overheads imposed by using the GASNet extended API rather than AM. We started with this fully portable GASNet implementation so we could evaluate the GASNet interface design and have a prototype implementation that runs anywhere.

[4] However, the traditional cautions about the possibility of deadlock in the presence of multiple locks still apply - the common solution is to define a total order on locks and always acquire them in a monotonically ascending sequence.

4.1 Methodology

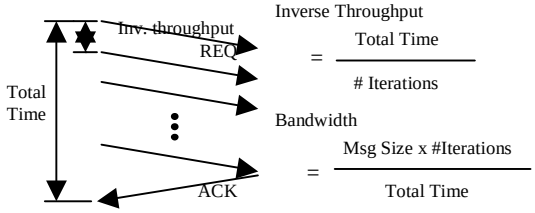
We ran two micro-benchmarks (a "ping-pong" test and a "flood" test) to evaluate the communication performance of GASNet in terms of round-trip latency time, bandwidth and message inverse throughput. The tests were all run between two nodes (an active sender and a passive receiver who is continually polling the network) and the test code is summarized in figures 4 and 5.

In the ping-pong test, the sender sends a get or put request and waits until the receiver acknowledges the request and the operation completes. Round-trip latency is the time from before the initial get/put request to after the operation is complete. A blocking get/put operation naturally represents the round-trip pattern, as it returns control to the client only after the acknowledgement has



Blocking	Non-blocking with explicit handle	Non-blocking with implicit handle
gasnet_get(…) gasnet_get(…) … gasnet_get(…)	h = gasnet_get_nb(…) gasnet_wait_syncnb(h) h = gasnet_get_nb(…) gasnet_wait_syncnb(h) … h = gasnet_get_nb(…) gasnet_wait_syncnb(h)	gasnet_get_nbi(…) gasnet_wait_syncnbi_all() gasnet_get_nbi(…) gasnet_wait_syncnbi_all() … gasnet_get_nbi(…) gasnet_wait_syncnbi_all()

Figure 4: Message and synchronization pattern for ping-pong test



Blocking	Non-blocking with explicit handle	Non-blocking with implicit handle
gasnet_get(…) gasnet_get(…) … gasnet_get(…)	h[0] = gasnet_get_nb(…) h[1] = gasnet_get_nb(…) … h[n-1] = gasnet_get_nb(…) gasnet_wait_syncnb(h, n)	gasnet_get_nbi(…) gasnet_get_nbi(…) … gasnet_get_nbi(…) gasnet_wait_syncnbi_all()

Figure 5: Message and synchronization pattern for flood test

been received. Non-blocking get/put operations are completed using an appropriate synchronization operation. We expect non-blocking operations performs to perform similarly to blocking operations in this test since they each entail a network round trip and the only potential difference is the software overhead on the sender. We test both variations of non-blocking operations (explicit handle and implicit handle) to fully explore the performance overheads.

In the flood test, the sender sends a large number of overlapped get/put requests and synchronizes all of them at the end with a single synchronization operation. This test is useful in revealing the maximum achievable bandwidth of the communication system (expected to be best for larger messages), and the inter-message issue time (inverse throughput) for small messages, an important performance characteristic for GAS languages. Note that we expect the bandwidth and inverse throughput reported for the blocking get/put operations in the flood test to be poor because the blocking functions don't allow any communication/communication overlap.

We amortized the timer overhead and granularity by running each communication test repeatedly (10,000 times) in a timed loop and dividing the total time by the number of iterations to calculate the average time per operation, as shown in figures 4 and 5.

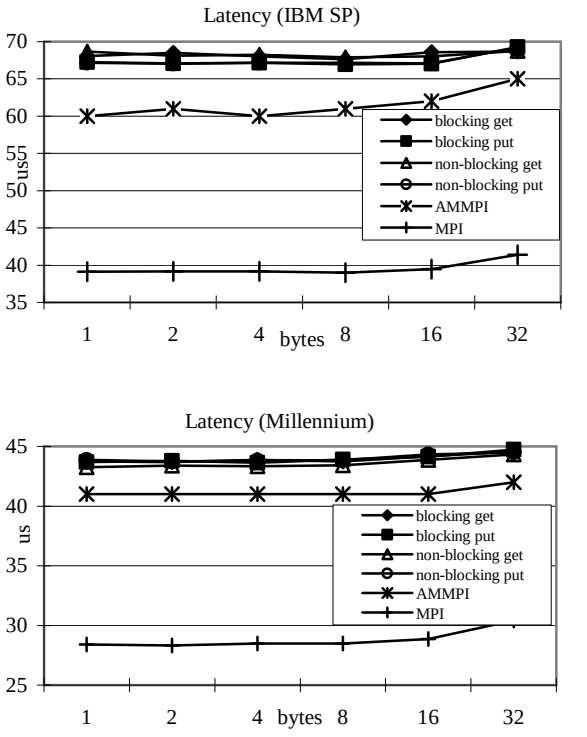


Figure 6: Round-trip ping-pong Latency

4.2 Operations measured

We measured the performance of various important GASNet extended API functions, including blocking transfers and non-blocking operations with explicit handles and implicit handles. We also measured the performance of AMMPI and MPI for reference since our prototype GASNet implementation is based on AMMPI (which runs over MPI).

Experiments showed that the two variations of non-blocking operations (explicit and implicit handle) performed nearly identically in all cases. The two variations provide different synchronization paradigms to the client code and are implemented somewhat differently in the extended API, but apparently the differences don't cause a noticeable change in performance - therefore we represent the two variations as a single entity in the following figures.

4.3 Latency Results

The ping-pong round-trip latency results are presented in table 1 and figure 6 (note the y-axis does not start at zero).

(us)	Blocking get	Blocking put	Non-blocking get	Non-blocking put	AMMPI	MPI
IBM SP	68.1	67.2	68.7	67.1	60.0	39.1
Millennium	43.7	43.7	43.3	43.9	41.0	28.4

Table 1: Round-trip Latency for 1-byte message

The round-trip latencies for all the extended API operations are very similar. This was expected because non-blocking operations are synchronized in the same way as blocking operations in the ping-pong test.

The GASNet gets and puts had a small additional overhead over AMMPI (around 7 us extra for the IBM SP and 2 us extra for the Millennium cluster). AMMPI had a much higher latency than MPI, and the gap between the two (around 21 us for the IBM SP and 13 us for the Millennium cluster) is due to the buffering that takes place in AMMPI as a direct result of the semantic mismatch between Active Messages and MPI.

4.4 Bandwidth Results

The flood bandwidth results are presented in table 2 and figure 7.

(MB/sec)	Blocking get	Blocking put	Non-blocking get	Non-blocking put	AMMPI	MPI
IBM SP	117.6	110.3	161.2	159.0	159.3	242.4

Table 2: Bandwidth for 128KB messages (network depth = 8)

For analysis purposes, we use the results from the IBM SP (the bandwidth results from Millennium had a large variance and are somewhat inexplicable).

The most important parameters dictating bandwidth performance are message size (in bytes) and network depth (aka queue depth - the number of outgoing messages that may be queued without blocking). Bandwidth generally increases with message size and we find it reaches a saturation point at around 128 KB. Finding the optimal network depth for a particular network requires some experimentation - if network depth is set too small, bandwidth drops quickly for large messages (due to a loss in communication/communication overlap which implies idle network cycles). Setting the network depth too high causes the bandwidth of small messages to drop due to increased cache miss penalties on the network buffers. We report results for a network depth of 8 on the IBM SP and 16 for Millennium.

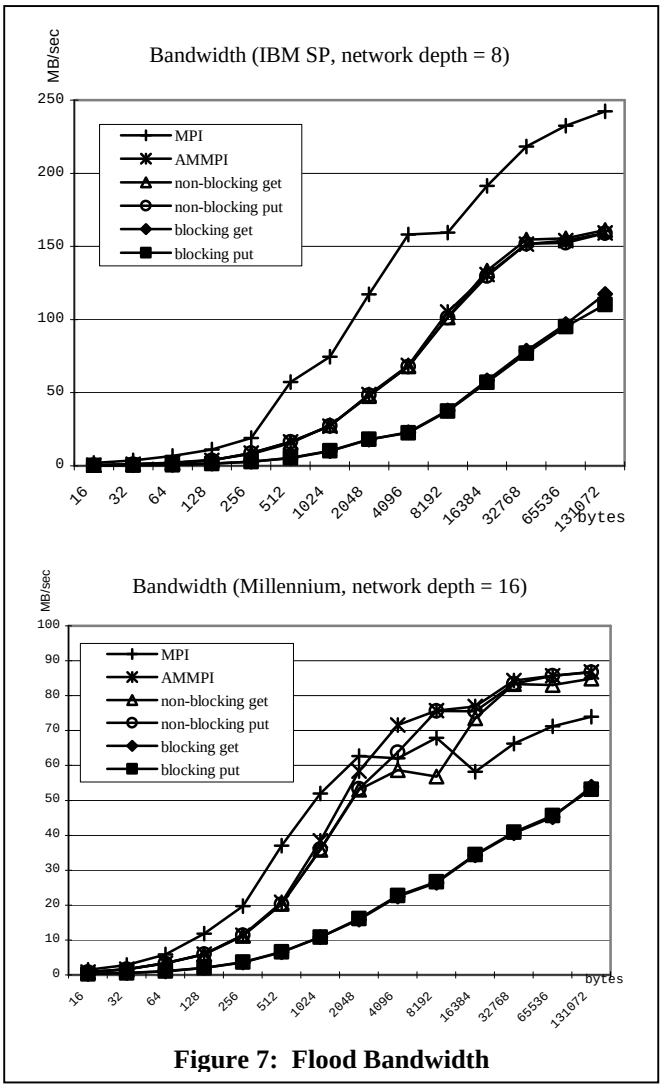


Figure 7: Flood Bandwidth

The bandwidth performance of the GASNet non-blocking operations closely matches the AMMPI bandwidth performance (e.g. about 160 MB/sec for 128 KB messages). The bandwidth graph also shows the expected benefits of non-blocking I/O in a flood test that allows communication/communication overlap (160 MB/sec vs. 114 MB/sec for 128 KB messages). As expected, the bandwidth of GASNet and AMMPI are still lower than that of MPI (242 MB/sec).

4.5 Inverse Throughput Results

The inverse throughput results are presented in table 3 and figure 8.

(us)		Blocking get	Blocking put	Non-blocking get	Non-blocking put	AM-MPI	MPI
SP	Inv. Throughput	78.5	78.6	28.8	28.9	28.9	7.6
	Latency	68.1	67.2	68.7	67.1	60.0	39.1
Mill	Inv. Throughput	56.3	55.8	18.2	17.8	17.7	10.7
	Latency	43.7	43.7	43.3	43.9	41.0	28.4

Table 3: Inverse Throughput for 1-byte messages

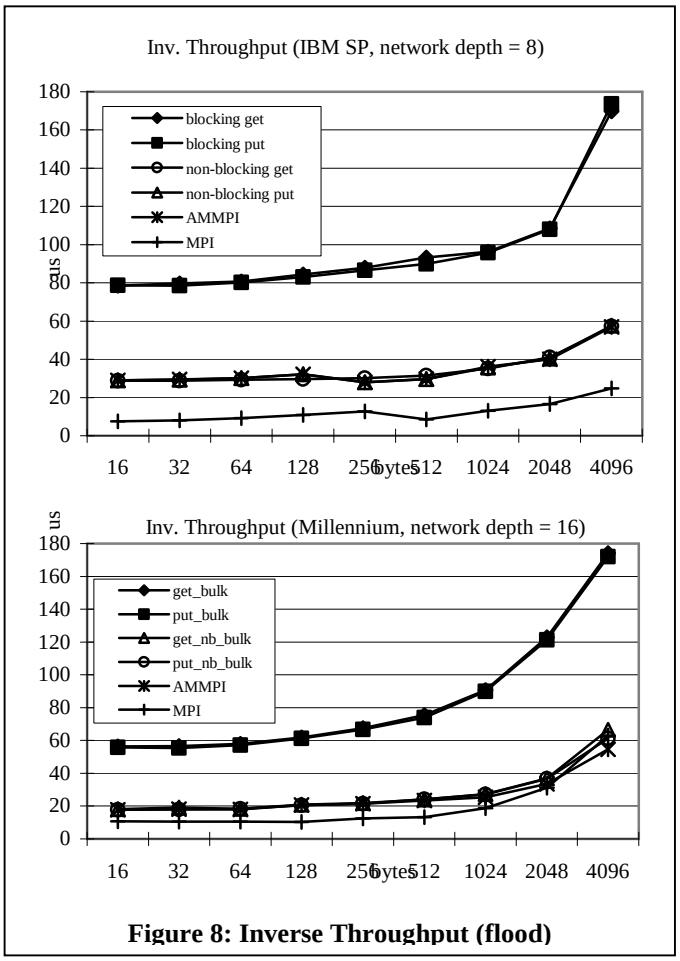


Figure 8: Inverse Throughput (flood)

The inverse throughput results show the GASNet non-blocking get/puts performed similarly to AMMPI but no better than MPI - much like the bandwidth results. These results confirm that non-blocking operations have higher bandwidth than blocking operations in a flood test because they can send requests (and therefore data) at a higher rate. The inverse throughput is an indication of inter-message issue time. Previous results [2] for MPI on these platforms show this issue time is dominated by

synchronous software overheads on the sender for small messages (rather than network gap). If we assume the message send overheads are similar on the replying node, we can subtract two inverse throughput times from the round-trip latency to discover that about 10 us on the SP and 8 us on Millennium are being spent outside the message send overheads. The previous results indicate that about half of these times are due to true, non-overlapped hardware wire latency, and the other half is due to message receive overhead which is not overlapped with anything else. Message send overheads are therefore the primary limiting factor in network performance on these systems. Previous results on the performance of the native GM layer on Myrinet indicate the send and receive software overheads are considerably lower than under MPI, which is compelling evidence that a future, native GM-based GASNet core implementation will perform considerably better than our AMMPI-based GASNet prototype and MPI itself (work is already underway on such a system to verify this claim).

4.6 Summary

Overall, GASNet performed much as we expected - specifically, the bandwidth and throughput performance were comparable to those of the underlying AM implementation. The round-trip latency was a little longer than that of AMMPI, but we believe this can be improved with further tuning. As expected, the absolute performance was lower than MPI and this is mostly attributable to AMMPI. Because GASNet exhibits low performance overhead over the underlying AM implementation, we expect that native implementations of the GASNet AM core will lead to excellent performance for the GASNet extended API operations.

5 Related Work

The language design of UPC was influenced by three previous research languages: Split-C [9], PCP [4] and AC [5], and is akin to other global-address space languages such as Titanium [30,25] and Co-Array Fortran [24,7].

Titanium is a parallel Java dialect that also supports a global address space memory model similar to UPC. The current Titanium compiler uses an implementation strategy similar to the NERSC UPC compiler, where programs are translated to intermediate C code and then compiled to machine code using a vendor-provided C compiler. Once our initial GASNet implementations are complete, we plan to add a GASNet backend to Titanium that will allow the language to also leverage future implementations of GASNet on various architectures. One of our goals in designing the GASNet interface was to provide a language-independent interface to make this reuse possible.

Our work on the GASNet core API was strongly influenced by Active Messages [17,28], and we carefully considered the capabilities of existing light-weight communication systems which are potential implementation targets for the GASNet API, such as IBM's LAPI [12], Myricom's GM [22] and Via/Infiniband [13,27].

MPI [20] is a well-established and standardized message-passing communication library that shares GASNet's goals of portability and high-performance in support of parallel programming. Unfortunately, there is a semantic mismatch between the one-sided get/put operations that are the critical primitives when implementing GAS languages and MPI's two-sided matched message send/recv operations, and this mismatch leads to performance degradation when attempting to implement one over the other. Additionally, MPI implementations have traditionally been tuned for high-bandwidth blocking message send/recv [2], whereas GAS languages typically require low-latency and low-overhead non-blocking operations. The MPI 2.0 standard [19] adds a "one-sided" communication interface that attempts to address some of these issues, but as discussed in [3] this interface has several problems that make it inadequate for the task of implementing GAS languages.

6 Conclusions and Future Work

GASNet is a powerful interface that provides portable, high-performance communication primitives useful in implementing global address-space languages. The two-level interface design allows rapid prototyping by implementing only the narrow GASNet core API, but allows further tuning through the direct implementation of selected operations in the extended API. Handler-Safe Locks provide safe (deadlock-free) explicit atomicity control for handlers, even with GASNet core implementations that run handlers using interrupts and/or concurrently on different threads. No-Interrupt Sections provide a cheap mechanism to prevent reentrancy errors in library functions called by handlers run by interrupt-driven GASNet core implementations. The absolute performance of the AMMPI-based GASNet prototype is not stellar (primarily because the MPI is not a good match for implementing global-address space languages – there is a semantic mismatch between one-sided, non-blocking get/put accesses and two-sided message send/recv). However, the relative performance of the GASNet

operations compared to AMMPI is promising, and we expect that GASNet will perform very well with a native GASNet core written and tuned for a particular network.

A good deal of work will take place on GASNet in the coming months as the NERSC UPC compiler development project progresses and we start expanding to various networks. We expect to implement GASNet natively on the following networks in the near future: IBM SP (LAPI), Myrinet (GM), Quadrics (ELAN), Cray T3E, Infiniband/VIA and possibly even Ethernet/UDP. We may also tune the AMMPI-based GASNet prototype implementation to achieve better performance on specific platforms (AMMPI has never been extensively tuned). As the compiler progresses, we also expect to augment the extended API with other useful operations, specifically collective communication support (e.g. broadcast, reduction, scan), and more sophisticated memory access operations (e.g. strided, scatter/gather). Finally, we plan to encourage other GAS languages to start using GASNet as a shared high-performance compilation target. If this effort is successful, we may even be able to convince high-performance network vendors to start providing GASNet implementations of their own, much like they currently do for MPI (a few vendors have already expressed an interest). This would help with our general goal of influencing network vendors to tune their hardware and software for low-latency, low-overhead communication rather than focusing primarily on peak bandwidth.

7 Bibliography

1. AMMPI home page http://www.cs.berkeley.edu/~bonachea/ammpi/index.html
2. Bell, Bonachea, et al. “An Evaluation of High-Performance Networks as a Compilation Target for Global Address-Space Languages”, *submitted to Super Computing 2002*.
3. Bonachea, “The Inadequacy of the MPI 2.0 One-sided Communication API for Implementing Parallel Global Address-Space Languages” http://www.cs.berkeley.edu/~bonachea/upc/mpi2.html
4. Brooks, E. D. III. “PCP: A Parallel Extension of C That is 99% Fat Free.” Technical Report UCRL-99673, Lawrence Livermore National Laboratory, Livermore, CA, 1988.
5. Carlson, William and Draper, Jesse, “Distributed Data Access in AC”, *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP*), July, 1995, p39-47.
6. Carlson, William. UPC f or the Cray T3E. http://www.super.org/upc/, ftp://ftp.super.org/pub/UPC/current/upctr.ps
7. Co-array Fortran web page. http://www.co-array.org/
8. Culler, D. et al., “Generic Active Message Interface Specification v1.1”, U.C. Berkeley Computer Science Technical Report, Feb 1995.
9. Culler, David et al. “Parallel Programming in Split-C”, *Proceedings of Supercomputing ’93*, Nov 1993, p262-273.
10. Dally et al., "Architecture of a message-driven processor". *Proceedings of the 14th Annual International Conference on Computer Architecture (ISCA)*, 1987.
11. El-Ghazawi, Tarek, Carlson, William and Draper, Jesse. “UPC Language Specifications, v1.0”, Feb 2001.
12. IBM LAPI web documentation http://www.research.ibm.com/actc/Opt_Lib/LAPI_Intro.htm
13. Infiniband web page http://www.infinibandta.org
14. Kuskin et al., "The Stanford FLASH multiprocessor", *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, 1994.
15. LBL-UC.Berkeley UPC Compiler Development Effort. http://upc.nersc.gov/
16. Mackenzie, Kubiatowicz, et al. “Exploiting Two-Case Delivery for Fast Protected Messaging”, *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1995.
17. Mainwaring, Alan and Culler, David “Active Messages: Organization and Applications Programming Interface.”, UC Berkeley Tech Report 1995. http://now.CS.Berkeley.EDU/Papers/Papers/am-spec.ps
18. Maquelin, Olivier, et al. "Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling", *the 23rd Annual International Symposium on Computer Architecture*, May 1996.
19. MPI 2.0 Specification http://www.mpi-forum.org/docs/mpi-20.ps
20. MPI 1.1 Specification http://www.mpi-forum.org/docs/mpi-11.ps
21. MuPC UPC Runtime System web page http://www.upc.mtu.edu/
22. Myricom, Inc. “The GM Message Passing System”, http://www.myri.com/scs/GM/doc/gm.pdf
23. Nieplocha, Jarek and Carpenter, Bryan. “ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems”, 1999.
24. Numrich, R.W. and J.K. Reid. “Co-Array Fortran for parallel programming.” *Fortran Forum*, volume 17, no 2, 1998.
25. Titanium web page http://Titanium.cs.berkeley.edu
26. UPC web page. http://upc.gwu.edu/
27. Virtual Interface Architecture Specification http://www.viarch.org/
28. von Eicken, Thorsten, Culler, David, et al. “Active Messages: a Mechanism for Integrated Communication and Computation”, 19th International Symposium on Computer Architecture, 1995.
29. von Eicken, Thorsten. “Active Messages: an Efficient Communication Architecture for Multiprocessors”, Ph.D Thesis, 1993.
30. Yelick, et al.”Titanium: A High-Performance Java Dialect”, *ACM 1998 Workshop on Java forHigh-Performance Network Computing*, Stanford, California, February 1998.

Appendix – The GASNet Specification

GASNet Specification, Version: 0.4
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.

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 "Global-Address Space Networking".

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.

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, but 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

Definitions:
==========
node - An OS-level process which called 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. Client 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)

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.

*** 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)

Implementors note:
* 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

Errors:
=======
Many GASNet core functions return 0 on success (GASNET_OK), or else they return errors from the following list, as specified by each entry point:
GASNET_OK = 0 no error
GASNET_ERR_RESOURCE_FAILURE
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 entry points will return GASNET_ERR_RESOURCE_FAILURE 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_init() 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.

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_t - the largest unsigned integer type that can fit entirely in a single CPU register for the current architecture and ABI. SIZEOF_GASNET_REGISTER_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_init()

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_MAXNODES
an integer representing the maximum number of nodes supported in a single GASNet job

GASNET_ALIGNED_SEGMENTS
defined to be 1 if gasnet_init() guarantees that the remote-access memory segment will be aligned at the same virtual address on all nodes. defined to 0 otherwise

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

```
================================================================================
                        ===== 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)

Job Control Interface

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

#define GASNET_SEGBASE_ANY ((void *)-1)
#define GASNET_SEGSIZE_EVERYTHING ((uintptr_t)-1)

int gasnet_init(int *argc, char ***argv,
                gasnet_handlerentry_t *table, int numentries,
                void *segbase, uintptr_t segsize,
               int allowFaults)
```

Called by all gasnet-based applications upon startup before any other processing of the command-line arguments takes place. Must be called before any calls to any other entry points 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. Initializes the GASNet system and performs any system-specific setup required, which may include spawning of parallel jobs. The semantics of any code executing before the call to gasnet_init() is implementation-defined (for example, it is undefined whether stdin/stdout/stderr are functional, or even how many nodes will run that code).
This call 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_FAILURE to indicate there was a problem acquiring the local requested 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_init().
May only be called once during a process lifetime, subsequent calls will return an error.
This call 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.

The handler table input (of size numentries) is 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 the table with function pointers and the desired handler index (or index 0 for "don't-care") - note that handlers 0..199 are reserved for GASNet internal use, and handlers 200..255 are available for client-provided handlers. Once gasnet_init() 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.

segbase and segsize are used to communicate the size and (optionally) the desired location of the shared 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 large-sized 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 the system page size). The client may provide a "hint" for the desired location of the segment by passing a pointer in segbase to a page-aligned address that points to a region of pages of the appropriate size (if no "hint" is to be provided, the client should pass segbase == GASNET_SEGBASE_ANY). GASNet is free to ignore the value of the hint and choose a different segment base address. The resulting segment assignment is guaranteed to be aligned on a system page boundary, and the values for all nodes can be queried using gasnet_getSegmentInfo().

The size of the segment is only limited by the size of the virtual memory, however some GASNet implementations will perform better when the size is less than some implementation-specific size. This implementation-specific size may be queried using gasnet_getMaxNativeSegmentSize().

If no hint is provided, 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 sbrk), and the function may fail if insufficient free _virtual_ pages can be acquired to accomodate the size request. If a hint is provided (and GASNet accepts the hint) then it is undefined whether the former contents of that memory (if any) are preserved. Clients who wish to map the entire virtual address space (including pages currently in use) should set the hint to zero and size to GASNET_SEGSIZE_EVERYTHING (and this is guaranteed to succeed, although it may provide suboptimal performance on some implementations).

If the implementation defines the macro GASNET_ALIGNED_SEGMENTS to 1, then gasnet_init() guarantees that the remote-access memory segment will be aligned at the same virtual address and have the same size on all nodes (and will fail if it cannot provide this, for example if different nodes request different sizes). Otherwise, this guarantee is not provided (although passing aligned "hints" _may_ still achieve the same effect).

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 large active message.

If allowFaults is zero, then GASNet guarantees that data transfer functions, large active messages and local accesses referencing these memory locations will succeed, even before any local activity takes place on those pages (i.e. in an implementation performing lazy registration, first touch = allocate). When allowFaults is non-zero, then such accesses to pages where the client has not taken system-specific actions to properly register the pages with the operating system _may_ cause a segmentation fault on some systems and/or implementations (this is most helpful when the segment size is GASNET_SEGSIZE_EVERYTHING, corresponding to the entire virtual address space).

Implementor's notes:
When allowFaults==0, 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.
Because the segment size is limited only by the virtual memory size, every implementation that pins pages needs a strategy for handling remote accesses 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 when allowFaults==0 (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.
Implementations which set GASNET_ALIGNED_SEGMENTS=1 and choose to accept the client "hint" need to check the hint pointers are aligned across processors and otherwise ignore some hints (may not be applicable to implementations which arrange to only run gasnet_init from a single node).

uintptr_t gasnet_getMaxNativeSegmentSize()

Retrieve the maximum memory segment size that may be provided to gasnet_init() which is still likely to provide the highest level of performance for the extended API data transfers and large request/reply active messages. Implementations with no maximum size (i.e. where performance is unaffected by segment size) should return GASNET_SEGSIZE_EVERYTHING. The return value of this function may depend on current system resource usage, but should return the same value for all nodes in a given job for the life of that job. The value returned is only a performance hint (which may be wrong) - the segment size selected by the client should never affect the correctness of the communication system.

This is the only function in the GASNet API which may be legally called before gasnet_init()

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 call it simultaneously with different exit codes, the result 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. If more than one thread calls gasnet_exit() within a given synchronization phase with different exitcode values, the value returned to the console will be one of the provided exit codes (chosen arbitrarily).

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 signals (e.g. segmentation fault) since some of these other signals may be meaningful (and non-fatal) to certain GASNet implementations.

Job Environment Queries
=======================

gasnet_node_t gasnet_mynode();

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

gasnet_node_t gasnet_nodes();

returns the number of nodes in the current GASNet job

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). The value of
numentries should be at least gasnet_nodes().
GASNet fills in the table with the registered segment base addresses and sizes for each node (including the local
one).
This is a non-collective operation.
Returns GASNET_OK on success.

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. The semantics of POSIX
setenv() are undefined in GASNet jobs (specifically, it will probably fail to propagate changes across nodes).

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 by 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", created 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_init(), 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_init() (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_init 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 to 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)

* how to handle 64-bit implementations? Need to specify...
  approach 1: make the handler args scale with pointer size (64-bit ints)
    cons: LP64 platforms (like Itanium) have 32-bit ints, but 64-bit ptrs
  approach 2: 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 blocks the current thread until ??????
```

* 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)

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. It is an error for a request handler to reply to any node other than 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 AMRequest*() to send a request to B
 it 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 AMRequest*() call.
  The request handler does some work on the arguments, and usually finishes by calling 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).
  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 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 includes 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.

Active Message Categories
==========
There are 3 categories of active messages:

Short - messages that carry a few integer arguments (up to gasnet_AMMaxShort())
  handler prototype:
     void handler(gasnet_token_t token,
                  gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);

Medium - messages that in addition to integer arguments can carry an opaque data payload (up to gasnet_AMMaxMedium() bytes in length)
  handler prototype:
     void handler(gasnet_token_t token,
                  void *buf, size_t nbytes,
                  gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);

Long - messages that in addition to integer arguments 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)
  handler prototype:
     void handler(gasnet_token_t token,

```
                  void *buf, size_t nbytes,
                  gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

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.

Active Message Size Limits

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,)

```
int 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().

```
int 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.

```
int 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.

```
int 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.

Active Message Request Functions

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

```
int gasnet_AMRequestShortM( gasnet_node_t dest,       /* destination node */
                            gasnet_handler_t handler, /* index into destination endpoint's handler table */
                            gasnet_handlerarg_t arg0, ..., gasnet_handlerarg_t argM--1 );
```

Send a short AM request to the given destination and handler, with the given M arguments.
gasnet_AMRequestShortM returns control to the calling thread of computation after sending the request message. Upon re-ceipt, the receiver invokes the appropriate active message request handler function with the M integer arguments.
Returns GASNET_OK on success.

```
int gasnet_AMRequestMediumM( gasnet_node_t dest,      /* destination node */
                             gasnet_handler_t handler, /* index into destination endpoint's handler table */
                             void *source_addr, size_t nbytes,   /* data payload */
                             gasnet_handlerarg_t arg0, ..., gasnet_handlerarg_t argM--1 );
```

Send a medium AM request to the given destination and 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().
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 re-ceipt, 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.

```
int gasnet_AMRequestLongM( gasnet_node_t dest,        /* destination node */
                           gasnet_handler_t handler, /* index into destination endpoint's handler table */
                           void *source_addr, size_t nbytes,   /* data payload */
                           void *dest_addr,                    /* data destination on destination node */
```

```
                            gasnet_handlerarg_t arg0, ..., gasnet_handlerarg_t argM--1 );
```

Send a long AM request to the given destination and 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_AMMaxLongRequest().
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_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 re-ceipt, 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.

```
int gasnet_AMRequestLongAsyncM( gasnet_node_t dest,        /* destination node */
                            gasnet_handler_t handler, /* index into destination endpoint's handler table */
                            void *source_addr, size_t nbytes,   /* data payload */
                            void *dest_addr,                    /* data destination on destination node */
                            gasnet_handlerarg_t arg0, ..., gasnet_handlerarg_t argM--1 );
```

gasnet_AMRequestLongAsyncM() has identical semantics to gasnet_AMRequestLongM(), except that the data payload source memory must NOT be modified until the matching reply handler has executed.
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 may block temporarily if the network is unable to immediately accept the new request.

Active Message Reply Functions
==============================

```
int gasnet_AMReplyShortM( gasnet_token_t token,       /* token provided on handler entry */
                            gasnet_handler_t handler, /* index into destination endpoint's handler table */
                            gasnet_handlerarg_t arg0, ..., gasnet_handlerarg_t argM--1 );
```

Send a short AM reply to the given handler on the requesting node (i.e. the one 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 re-ceipt, the receiver invokes the appropriate active message reply handler function with the M integer arguments.
Returns GASNET_OK on success.

```
int gasnet_AMReplyMediumM( gasnet_token_t token,       /* token provided on handler entry */
                            gasnet_handler_t handler, /* index into destination endpoint's handler table */
                            void *source_addr, size_t nbytes,   /* data payload */
                            gasnet_handlerarg_t arg0, ..., gasnet_handlerarg_t argM--1 );
```

Send a medium AM reply to the given handler on the requesting node (i.e. the one 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().
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 re-ceipt, 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.

```
int gasnet_AMReplyLongM( gasnet_token_t token,       /* token provided on handler entry */
                            gasnet_handler_t handler, /* index into destination endpoint's handler table */
                            void *source_addr, size_t nbytes,   /* data payload */
                            void *dest_addr,                    /* data destination on destination node */
                            gasnet_handlerarg_t arg0, ..., gasnet_handlerarg_t argM--1 );
```

Send a long AM reply to the given handler on the requesting node (i.e. the one 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().
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 re-ceipt, 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.

Misc. Active Message Functions
==============================

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 may be a no-op on some implementations (e.g. purely interrupt-based implementations).
Returns GASNET_OK unless an error condition was detected.

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.

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.

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 Richard Stevens' "Advanced Programming in the Unix Environment", p 278). 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":

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.

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.

* 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

* 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 by called within the no-interrupt section are:
  gasnet_mynode(), gasnet_nodes(), gasnet_hsl_lock(), gasnet_hsl_unlock(), gasnet_AMReply*()

* gasnet_hold_interrupts() and gasnet_resume_interrupts() should not be called from within a handler context - handlers are run within an implicit no-interrupt section

* 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
 Strictly polling-based implementations which never interrupt a thread can implement these as a no-op.

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).

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.

```
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.

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

Lock and unlock HSL's. gasnet_hsl_lock() will block until the 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_unlock() releases the lock previously acquired using gasnet_hsl_lock().
It is erroneous to call these functions on HSL's which have not been properly initialized.

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 above). gasnet_hold_interrupts() and gasnet_resume_interrupts() must not be explicitly called while holding an HSL

* HSL's may _not_ be locked recursively (i.e. calling gasnet_hsl_lock() 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.

* Any handler which locks one or more HSL's MUST unlock them all before exiting or calling gasnet_AMReply*()

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 when GASNET_SEQ 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.

==========
===== 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.

Memory-to-memory data transfer functions:
==========
These comments apply to all put/get functions:

* 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_init()), 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

Blocking memory-to-memory transfers
==========

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 and 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

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.

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

Blocking operation that executes memset(dest, val, nbytes) on the given node (and has the same semantics as that function).

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 (put or get) and a subsequent synchronization on the completion of that operation before the result is guaranteed.

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)

There are two categories of non-blocking operations:
"explicit handle" (nb) - return a specific handle to caller which is used for synchronization
this handle can be used to synchronize a specific subset of the nb operations in-flight
"implicit handle" (nbi) - don't return a handle - synchronization is accomplished
by calling a synchronization routine that synchronizes all outstanding nbi operations

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 an unlimited number of non-blocking operations in-progress - that is, the client is free to issue an unlimited number of non-blocking operations before issuing a sync operation, and the implementation must handle this correctly without deadlock or livelock.

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 type whose contents are implementation-defined, with one exception - every implementation must provide a 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.

```
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 a explicit 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.

```
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).

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

Non-blocking operation that executes memset(dest, val, nbytes) on the given node (and has the same semantics as that function). 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).

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.

```
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.

```
void gasnet_wait_syncnb_all(gasnet_handle_t *, int numhandles)
int  gasnet_try_syncnb_all (gasnet_handle_t *, int 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.

```
void gasnet_wait_syncnb_some(gasnet_handle_t *, int numhandles)
int  gasnet_try_syncnb_some (gasnet_handle_t *, int numhandles)
```

These operate analogously to the 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.

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

```
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)
```

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.

Synchronization for implicit-handle non-blocking operations:
==========
Synchronize on the outstanding 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).

```
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.

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.

```
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 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 associated with this access region
  while (...) {
    gasnet_put_nbi_shared(...); // becomes associated with this access region
  }

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

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

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

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

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.

Value Put
=========
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

Blocking Value Get
==================

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

Blocking value get - this function returns the fetched value to avoid forcing incoming values to local memory in
generated code (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 nbyte integral value on the current architecture and setting the high-order bits (if any) to zero
(i.e. no sign-extension)

Non-Blocking Value Get (explicit-handle)
========================================
This operates similarly to the blocking form of value get, but is split-phase
split-phase value gets are synchronized independently of all other operations in gasnet

typedef ??? gasnet_valget_handle_t;

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

Barriers:
=========
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()).

void gasnet_barrier_notify(int id)

Execute the notification for a split-phase barrier, with a barrier value
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 id == GASNET_ANONYMOUS_BARRIER then the barrier is anonymous and has no specific id.

int gasnet_barrier_wait(int id)
```

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 the supplied barrierval doesn't match the value provided in the preceding gasnet_barrier_notify() call made by this node or any other node in this synchronization phase.
Otherwise, returns GASNET_OK to indicate that all nodes have called a matching gasnet_barrier_notify() and the barrier phase is complete.

int gasnet_barrier_try(int id)

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 barrierval, 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

#define GASNET_BEGIN_FUNCTION() ???

This macro may _optionally_ be placed at the top of functions which make calls to the extended API. 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 the function (before any declarations or calls to the API in that function).