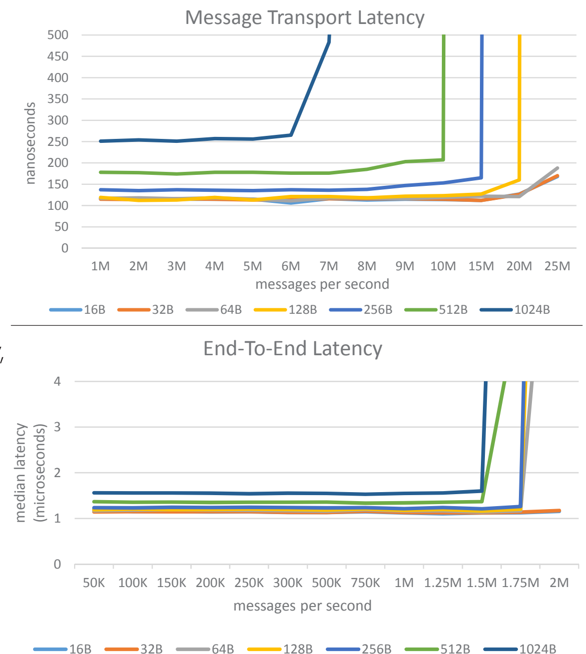
TECHNICAL

FAST MESSAGE ENRICHMENT

Low-Latency Algorithmic Message Processing

Performance With a Purpose

The Advanced Message Processing System (AMPS), from 60East Technologies, is designed for the most demanding real-world applications. Our engineers specialize in understanding technological innovations and using those innovations to deliver consistent low-latency and high performance throughout an entire system. Multi-core processors make new levels of performance possible. This briefing shows how you can use AMPS and shared memory to accelerate message enrichment in your high-throughput, low-latency applications.



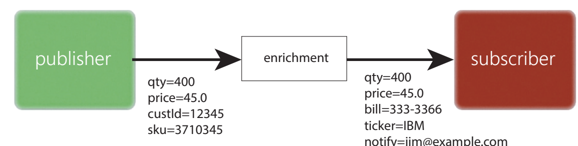
Shared Memory Messaging

Shared memory allows AMPS to move data between processes at sub-microsecond latency. As the name suggests, the two processes work with a common area of memory. In an ideal state, the processes run on the same core, or cores in the same physical socket, so they are truly sharing memory. However, shared memory works well even when sharing data across processor sockets. Message transmission involves synchronization and copying memory between cores or between sockets, operations highly optimized at the hardware level. These copies have very low latency, as shown by the Message Transport Latency graph, which measures the time to publish a message from a client to AMPS. The AMPS engine helps keep latency low through the entire system. The End to End Processing Time graph shows latency from a shared memory client through AMPS to another shared memory client. AMPS latency scales smoothly with message size, and AMPS maintains consistent latency until message throughput saturates the processing power of a single CPU.

Context is King

The numbers are impressive, but the test is how well real applications meet business needs. Most applications use a combination of rapidly-changing information and more stable information. In an order processing application, the active orders, prices and quantites vary millisecond by millisecond. Customer account numbers, ticker symbols, and catalog numbers are relatively stable. To act on the quickly changing data, applications *enrich* messages—add additional data needed for processing—as part of the message processing pipeline. The challenge is to enrich messages while keeping latency low in the overall system. Performance in one part of the system doesn't help if another part of the system can't keep up.

Some applications require each client to enrich the message. This can cause problems when each client has a different "version of the truth", and can also cause performance problems when clients need to update the reference data. This may be the only option for systems that use multi-cast messages or direct delivery to message processors. AMPS includes sophisticated filtering and routing, so systems built with AMPS often create enrichment processors that serve as a coherent point of enrichment.

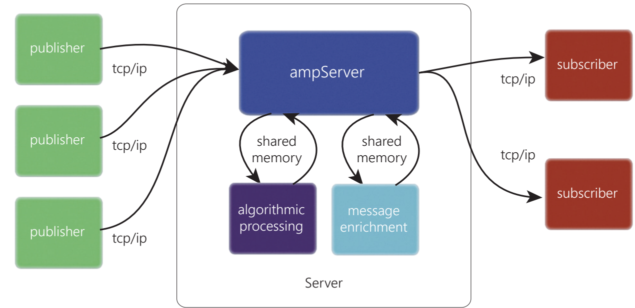


Low-Latency Processing

For the most flexible message processing, you create a client program based on the AMPS API that receives a message, enhances it, and then publishes it back to AMPS. To make this easy, there's a simple pattern: messages include a field that says whether a particular processor has enhanced the message. Each processing program uses filters using that field — for example, the processor requests messages where `/myStepDone = false`. Once processing is done, the processor republishes the

message to the same topic with `/myStepDone = true`. Subscribers use the filter `/myStepDone = true`, and only receive processed messages.

Creating a shared-memory external processor is simple. You use the AMPS C or C++ client libraries and change the server address to use the shared memory protocol rather than TCP/IP. You then update the configuration on the AMPS server to enable shared memory. There are no other changes required to your code.



To get the best performance from shared memory, pay attention to the CPU usage of the AMPS server and your application. In our labs, latency increases by an order of magnitude if the CPU is saturated. We recommend using `numactl` to bind the message processors to the same NUMA node as the core AMPS functions if that node consistently uses less than 100% of the CPU capacity with these processes on the same node. If the processors and AMPS will use the CPU capacity of the node, move processors to another NUMA node so that the CPU capacity is consistently under 100% on all nodes. If the message processors use all of the CPU capacity on the machine, we recommend moving the lowest-priority processor to another machine and switching it TCP/IP.

Simple to Code

With AMPS, creating a shared memory processor is just as easy as creating a simple publisher or subscriber. The AMPS client library handles all of the details of managing shared memory, including minimizing copies, polling, flow control, and so on. Your code simply receives the messages and acts on them.

```
void MessageHandler(const Message& message, void* userData) {

  try {

    // The subscribe call passes an AMPS client as the userData
    // to the function. Retrieve the client.

    AMPS::Client& client = * static_cast<AMPS::Client*>(userData);

    // Call a function that processes the incoming message and
    // returns an outgoing message. We use std::string here for clarity.
    // A production system would use a more optimized approach.

    std::string outMsg = parseAndProcess(message.getData());

    // Publish the message to an outgoing topic, using the provided client.

    client.publish("orders", outMsg);
  }
  catch (const AMPS::AMPSException& e) {
       std::cerr << e.ToString() << std::endl;
       exit(1);
  }
}
```

In-memory message processing provides all of the features of the AMPS high performance engine, including filtering by topic and content.

The sample here shows a simple message handler. AMPS calls this message handler for each message received. The handler processes the message and then sends the processed message back to AMPS. This is the basic receive and publish loop common to all algorithmic processors in AMPS. Notice that your processor does not need to deal with handling the shared memory. AMPS takes care of those details for you.

With the AMPS client libraries, It's never been simpler to create high-performance algorithmic processing.

Built to Scale, Built to Last

AMPS is in use in some of the most demanding applications in the financial industry. AMPS instances process billions of mission-critical messages every day. When a single delayed or missed message means business disruption, you can rely on AMPS to do the job.

Staffed by veterans of companies such as Morgan Stanley, Bank of America/Merrill Lynch, IBM, Microsoft, and Rogue Wave Software, the 60East team provides unparalleled support with direct access to the development team.

For more information, visit our website or contact sales@crankuptheamps.com.