Astronomical Data Analysis Software and Systems XII
ASP Conference Series, Vol. 295, 2003
H. E. Payne, R. I. Jedrzejewski, and R. N. Hook, eds.

Rapid Development for Distributed Computing, with Implications for the Virtual Observatory

Michael S. Noble

Center for Space Research, Massachusetts Institute of Technology

Abstract. There remains a significant gap in grid and distributed computing between what *can be achieved* by advanced research groups and what *is realized* by typical scientists on the desktop. We argue that Java-based tuplespaces (JTS) can shrink this gap by lessening the impediments of complexity and institutional buy-in which traditionally accompany the development and use of distributed systems. Drawing from our benchmarking experience, we analyze the strengths of JTS for node configuration, scalability, and failure recovery. We then illustrate how JTS were leveraged to rapidly prototype a powerful end-to-end Virtual Observatory analysis thread within the Chandra Data Analysis System.

1. Introduction

As has been charted by Top500.org for the past decade, peak computer performance continues to grow at an impressive clip. However, these advances in parallel and distributed computing tend to be accessible to only a small portion of researchers, mainly those within focused research projects at national, university, and corporate laboratories. These groups characteristically benefit from generous funding and can afford to purchase (or build) the best equipment and hire whatever staff is necessary to program and maintain it.

At the other end of the performance spectrum are individual researchers and small collaborative groups. These are by far the more common organizational units in which science is daily practiced, and cumulatively represent a much larger fraction of the total researcher population. Yet the scale of computing power available to many of these researchers, or perhaps more accurately that which is *actually used by them on a regular basis*, remains merely the desktop workstation.

A variety of issues inhibit the adoption of high performance computing (HPC) by the individual, many of which can be grouped under the rubric of *the buy-in problem*. After several decades of research it is generally accepted that parallel and distributed programming has not kept pace with improvements in hardware (Hwang & Xu 1998). Despite the breadth of programming models available, HPC has not evolved into the mainstream, and skilled programmers are still hard to find. The complexity of industrial-strength HPC toolkits typically requires a significant investment of time and learning, not only for application or algorithm development, but also for installation and maintenance, especially on today's heterogeneous networks. Our experience is that few

practicing scientists have the time or inclination to cultivate these skills (nor, as can be the case, do their technical support staff), with the net effect being that much of the computing potential available to them, at least in principle, goes unused.

This trend manifests itself at several levels. The clearest indication is at the granularity of the network, where the desire to harness the CPU cycles of idle machines has motivated an entire subdiscipline of computer science. The second indication stems from the cycles of hardware upgrades which play out at every computing center. In step with Moore's Law, our older machines are replaced with newer ones, with few second thoughts as to their ultimate destination; as often as not these otherwise functional systems wind up collecting dust in a dark closet. Lastly, observe that while we are at the cusp of another evolutionary step in desktop computing, namely the emergence of the personal multiprocessor system, the extra CPUs in such machines are as yet rarely employed in general scientific development, particularly within the analysis packages used most frequently by astronomers. The wasted CPU cycles in each of these cases would in principle be straightforward to reclaim if parallel and distributed computing skills were more commonplace.

Together these factors create a formidable entry barrier for HPC newcomers, a class into which—drawing from our collaborations at several international research centers—a majority of astronomers still fall, and strongly motivate our investigation of tuplespaces (TS) for rapid distributed computing development.

2. Tuplespaces Primer

Introduced in the 1980's, the Linda model has been thoroughly described in the literature and exists in a variety of implementations. The central idea is that of *generative communication* (Gelernter 1985), whereby processes do not communicate directly, but rather by adding and removing tuples (ordered collections of data and/or code) to and from a *space* (a process-independent storage abstraction with shared-memory-like characteristics, accessed by associative lookup). This provides an object-like form of distributed shared memory whose semantics can be expressed by adding only a handful of tuple operators [essentially write(), read(), take(), and eval()] to a base language. This approach gives rise to distinctive properties such as time-uncoupled anonymous communication (sender and receiver need not know each other's identity, nor need they exist simultaneously), networked variable sharing (tuples may be viewed as distributed semaphores), and spontaneous multiparty communication patterns that are not feasible in point-to-point models. The simplicity and clean semantics of TS present an attractive alternative to other parallel programming models (e.g., message-passing, shared-variable, remote-procedure call) and foster natural expressions of problems otherwise awkward or difficult to parallelize (Carriero & Gelernter 1988).

2.1. Commoditized by Java

Linda tuplespaces have been offered as custom C, C++, and Fortran compilers, with a commercial license that can be a barrier to adoption for smaller research groups. Furthermore, as is the case with other parallelizing compilers, and with

message passing libraries, such Linda programs are architecture-specific binary code, meaning one cannot compile a *single* Linda application for use on highly heterogeneous networks.

The marriage of the TS model with the popular Java platform addresses these concerns, and extends Linda with transactions, tuple leasing, and event notification. The arrival of two free, commercial-grade Java tuplespace (JTS) platforms—Sun's JavaSpaces (Freeman et al. 1999) and IBM's TSpaces (Wyckoff et al. 1998)—opens an entirely new avenue of exploration in distributed computing with commodity technology.

2.2. JTS for Science: The Perfect versus The Good

The notion of using Java for computational science is not new, and significant advances have been reported in numerous areas (JavaGrande Forum[1]). Given the tight deadlines under which research is conducted, the value of using commodity mechanisms is clear. Observational astronomers, for example, who cannot analyze data with the software at their disposal typically have little choice but to write their own if they wish to publish during their proprietary period. Theorists may be impacted, too, considering e.g., the role high-performance simulations play in development and confirmation.

It is our contention that, if asked to choose between achieving *theoretically maximum* performance some unspecified N months into the future or achieving *suboptimal but good* performance in a fraction of that time, many researchers would gladly select the latter. In Noble & Zlateva (2001) we discuss JTS in this context and present benchmark results which exhibit good speedups for several parametric master/worker algorithms, relative to both sequential Java and native compiled C (Figure 1). Other researchers have also reported substantial

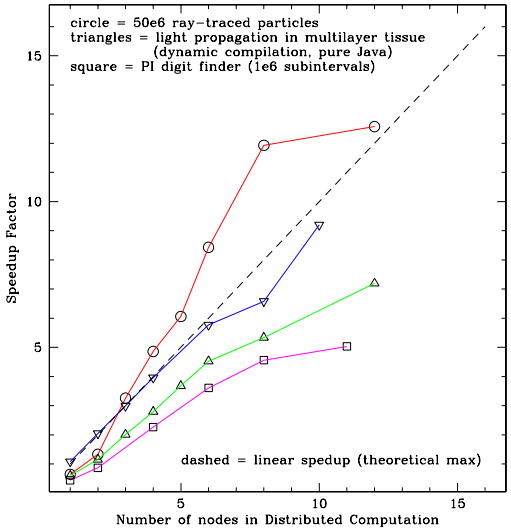


Figure 1. Peak JTS speedups vs. sequential GCC -O3

[1] `http://www.javagrande.org`

speedups using JTS (Batheja & Parashar 2001, Teo et al. 2002). While several weaknesses of JTS were noted in our work (notably communication latency and a lack of collective broadcast/gather/reduce communication operations), they nonetheless remain enticing as a poor man's HPC platform by addressing off-the-shelf numerous issues with distributed systems.

2.3. Generality

The programmatic simplicity of tuple retrieval in Java masks a powerful network-enabled runtime code loader. This mechanism fosters an uncoupling of generic "dumb workers" (which loop endlessly, waiting to pull tasks from the space to blindly execute their *run()* methods) from task execution code, and provides several clear advantages over traditional message-passing or shared-memory HPC implementations.

First, by remaining wholly ignorant of the details of the computations they perform, generic JTS workers may be written just once for a variety of computations, and breezily deployed on any Java-capable platform. While it is true that similar levels of runtime generality and portability are attainable by other means, e.g., Java RMI, most fall short of the ease of use and semantic clarity that is *built in* to JTS in that the programmer can be required e.g., to abstract the network interactions themselves (or use a package which does so), use RPC compilers to explicitly generate client stubs, and so forth.

Second, such workers need be started only once *per compute node*, after which they idle as low-consumption daemons until tasks appear in the space. With native-compiled PVM, MPI, or OpenMP implementations, however, master and worker tend to be far more tightly coupled. This approach is typically characterized by workers being coded with explicit algorithmic knowledge and masters being directly responsible for the spawning of workers. Together these make the typical worker unsuitable for use in a wide variety of computations, and present additional difficulties on heterogeneous networks as one body of compiled code cannot be used on multiple architectures.

2.4. Dynamic Scaling

Likewise, when masters are responsible for spawning workers en masse the size of the worker pool is usually fixed in advance, effectively prohibiting the addition of workers after the computation has initiated. With static scaling, long-running calculations, for example, cannot be sped up after launching, and instead need to be restarted to add more horsepower. In a JTS solution the master—whose chief responsibilities are to parcel a job into subtasks, funnel them to the space, and collect results—directly interacts with only the space, which allows the computation to be *dynamically scaled*, either up or down, simply by adding or removing workers. Moreover, given the simplicity of threading in Java relative e.g., to POSIX, it was straightforward to have workers in our benchmark framework adapt to multiprocessor hosts by spawning multiple lightweight-process copies of themselves.

It is worthwhile to note that using N=1 worker is effectively a sequential invocation, while parallelism is achieved by using N=k>1 workers. This semantic clarity can be very beneficial to the application developer, since one of the more difficult aspects of programming for concurrency is that sequential algorithms

frequently do not map cleanly to parallel implementations. In such cases it is difficult or impossible to utilize one body of sourcecode for both sequential and parallel execution.

2.5. Factoring Administrators Out Of The HPC Equation

As is the case with large-scale databases, many traditional approaches to HPC require direct involvement of a system administrator, and can eventually mandate hiring additional, even dedicated, staff. Whether to research and purchase exotic massively-parallel hardware, install enabling software on multiple nodes, activate low-level daemons with root privileges, create special user accounts, or manage pooled disk space, the need to involve yet another outside party diminishes individual autonomy and introduces a recurring usage bottleneck.

Recent work, though, shows that combining network class loaders with JTS can eliminate many of these difficulties (Noble 2000,[2] Lehman et al. 2001, Hawick & James 2001). In fact, in the 47-node MAN testbed used to derive the results of Figure 1: the JTS system software was installed on non-privileged diskspace, and was booted from an ordinary user account, master and workers were run from ordinary user accounts, and 6 of the 14 machines used did not even have a local JTS installation.

Remote Node Configuration In fairness we should note that some of these benefits were derived as much from our lack of security concerns as from any specific technology choice, but our contention, again, is that to promote rapid development and results "faster and looser" is the prevailing mindset in smaller research groups (especially those whose goal is *not* publication of research papers in computer science). However, the fact that a JTS installation was *not required at all* on participant nodes is considerable step forward in the configuration and management of distributed systems on heterogenous networks. To achieve this we merged the client portion of JavaSpaces[3] and our worker code into a single bootstrap jar file, and made it available on a well-known compute portal.

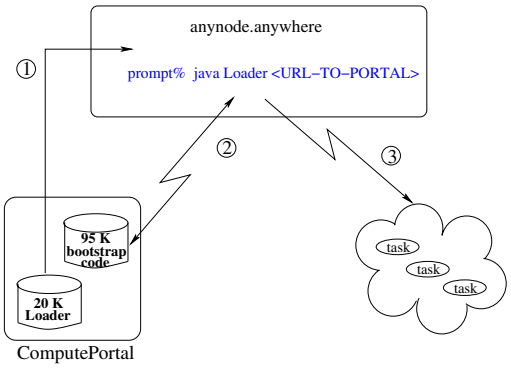


Figure 2. Simplified remote node configuration with JTS.

[2] `http://hea-www.harvard.edu/~mnoble/tonic/doc`

[3] We have not tried this yet with TSpaces, but (Lehman et al. 2001) suggests it will work.

Users electing to participate in the distributed computation framework need only install a network launcher onto the prospective node(s), the cost of which can be as small as a 20K download.[4] At runtime the loader is pointed at the established portal to retrieve the bootstrap jar and invoke the worker class within, after which the worker contacts the space and idles for compute tasks. Whatever class loader is used, it need be downloaded just once per node, after which it can be reused regardless of how many revisions are made to the bootstrap code or computation framework.

With this approach it is no longer necessary to consume administrator time to configure or employ distributed systems—a JTS solution may be installed and activated without knowledge of the system password *on any machine*; nor is it necessary to maintain multiple native binary builds—of either the HPC toolkit or your computational codes—to utilize multiple CPU architectures in a single distributed computation. The significance of this technique increases in direct proportion to both the number and architectural diversity of nodes within the computation environment, as well as the workload of your system administrators.

2.6. Fault Tolerance and Checkpointing

In sharp contrast with other parallel and distributed programming mechanisms, failure recovery is an intrinsic feature of available JTS implementations. First note that any space—a persistent external store into which tasks are doled out and intermediate results accumulate—is itself a checkpointing mechanism. Checkpointing, where partial results are incrementally saved, is used in long-running calculations to avoid restarting from scratch after failures. It contributes nothing to computed solutions, per se, but typically requires explicit coding within the algorithm for message-passing and shared-memory implementations. Second, while in our benchmarking efforts we found the JTS implementations very robust (capable of running unhindered for weeks and months on end), it is a fact of life that node failures will occur. With JTS transactions, though, the demise of participants *during* a computation is in principle fully recoverable. Wrapping worker interactions with the space within a transaction, for example, ensures that if a worker dies the transaction cannot commit; this in turn will cause the subtask being computed by the worker to be automatically returned to the space (in virgin form) as the transaction is unrolled. Similar benefits may be gained by wrapping the results collection phase (in the master) within a transaction.

3. Prototyping the Virtual Observatory

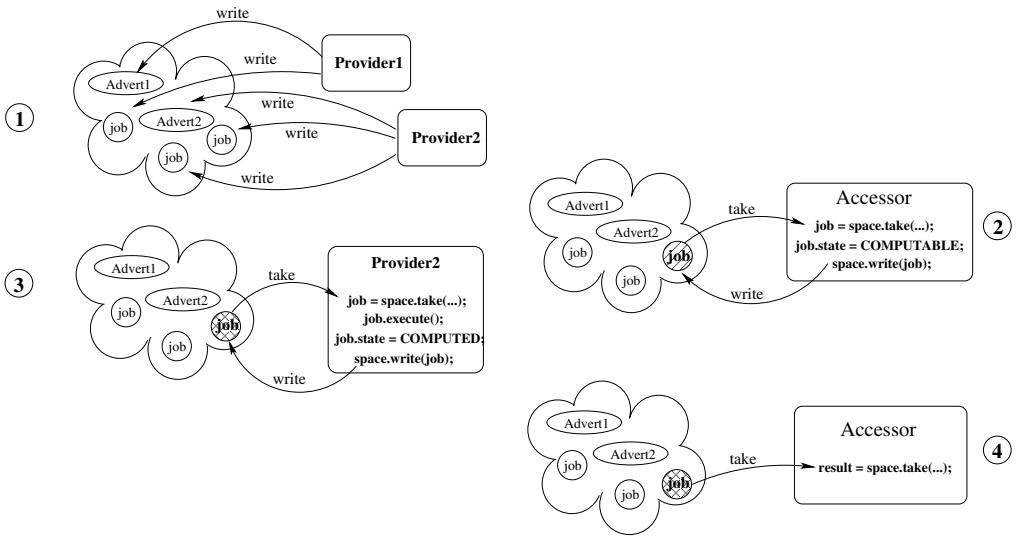
A number of the benefits discussed above map well to the Virtual Observatory problem space. The loosely coupled master/worker pattern, for example, generalizes to the use of TS for coordinating ensembles of anonymous processes (Gelernter & Carriero 1992). A fully-realized VO should in principle permit the

[4]Our experiments were conducted in 2000, prior to the wide availability of JNLP and Java WebStart. Instead, we implemented a custom loader based upon WebRun from Michael Sinz.

spontaneous formation of any number of such ensembles. The degree to which this becomes accessible to the typical astronomer, unencumbered by system administrator or programmer hand-holding, will be a key metric of success. Other crucial factors include the flexibility with which back-end services can be glued together, how rapidly they may be deployed, and how well VO mechanisms integrate with existing astronomy software. To explore these concepts further we have constructed a testbed (Cresitello-Dittmar et al. 2003) which uses JTS to define and coordinate a number of services (name resolution, multi-archive lookup and retrieval) which automate the generation of composite images and multi-waveband flux plots. Several distinguishing features of the prototype are that it is accessible from a publicly-available analysis toolset (CIAO), that the VO capability is *anonymously* invoked, and that it was deployable after a very short coding period by two new hires. User interactions are managed by a scriptable Gtk GUI coded in S-Lang and loaded at runtime into the *chips* visualization tool. The name and identity of each VO service was discovered at runtime by the GUI client, which needed knowledge only of how to contact the space.

3.1. Testbed Architecture

Workflow is modeled in our testbed by Service Providers, Requestors, and Portals, which interact by exchanging Advertisement and Job tuples. An Advertisement describes *what* work the provider is willing to perform, while a Job—a quanta of work—indicates *how much*. In the following illustration two providers have written two advertisements and three jobs in an INITIAL state. By writing



two job tuples to the portal Provider2 indicates that it is willing to service two requests for Advertisement2 simultaneously, perhaps because it is a multiprocessor (the owner of, say, a 16-cpu machine might choose to make any number of its processors available in like fashion). The accessor, having scanned the portal for advertisements, requests that a service be computed on its behalf by retrieving a job from the space, marking its state as COMPUTABLE, and writing it back

to the space. Provider2, who is woken by the presence of a computable service request, executes the job, marks its state as COMPUTED, and writes it back to the space. Finally, the accessor notices the arrival of its completed request and retrieves the result.

3.2. Analysis

This architecture is heavily influenced by the decoupled nature of TS communication, and manifests a number of useful properties beyond mere clarity of object flow. For example, advertisements gracefully expire from the portal if not refreshed by the provider within a lease period, which serves as an implicit heartbeat mechanism. With such *dynamic service lifetime management*, partial state—which can accumulate in an explicit deregistration model when a provider joins the portal and then suffers a catastrophic failure—is automatically flushed. Assuming zero transmission errors, a requestor is guaranteed—within the granularity of the lease—that its service request will at least initiate, rather than merely spilling into the ether of an offline provider.

Likewise, the separation of advertisement from job execution yields an implicit mechanism for detecting a saturated provider *without further contributing to that saturation*: if an advertisement exists in the portal with no corresponding job tuples then the provider must be busy servicing other requests. A client can detect this state without issuing an ill-fated request (yet more work for the overloaded provider) doomed to merely timeout. This fosters smoother client operation, especially important when human interaction is involved, and promotes deterministic behavior of the system as a whole. The use of jobs as quanta of work lets providers retain full control over how much community service they will perform, and when, and makes it *virtually impossible* for them to saturate from requests. While the space itself is a potential saturation point, the tasks it handles (mostly funneling small-ish tuples back and forth) are not computationally expensive, and its function can be replicated for scalability as needs arise (although more investigation is needed in this area).

Another form of load balancing is manifest in the ability of the space to store partial results. Imagine for instance a provider who offers a computationally expensive service that takes N input parameters. Since the result of that computation can be stored in the space (or a reference to it if the result, e.g., a file, is too large), new requests with matching inputs need not result in entirely new service invocations at the provider.

Finally, note that while service discovery and requests are coordinated by the space, its use is not mandatory. A provider is free to negotiate more effective means of interaction with the requestor, and in fact archive retrievals in our testbed are carried out by direct download between the hosting site and client.

What About Web Services? Web Services advocates in the VO community may contend that a Java-centric approach such as we have employed here lacks neutrality by binding one too closely to a specific language or object model. While this position is not without merit, its wholesale acceptance would seem to require overlooking several issues of significance.

For one, many of the features described here—simple service coordination, accumulation and persistence of partial results, asynchrony, transactional sup-

port, fault tolerance, and dynamic lifetime management—are not well-supported in other service hosting implementations, such as those providing Web Services through a UDDI registry. These factors—plus the heavy bias of UDDI towards commerce, and the performance of SOAP (Davis & Parashar 2002, Govindaraju et al. 2000[5])—should concern those considering adopting Web Services for scientific use. In response the grid community is developing more robust alternatives, such as the Open Grid Services Architecture (forming the core of the next generation Globus 3.0 Toolkit, which incidentally also requires Java for service management, though wider language support is planned) and the Web Services Discovery Architecture (WSDA). Until these mature, however, the scientific community is left with few off-the-shelf options for service management. Our JTS-based scheme productively fills this interim niche.

Next, a commitment to use Java *somewhere* within a project does not mandate its use *everywhere*, given the availability of the Java native interface (JNI) and fork()/exec(). Our testbed amply demonstrates this by seamlessly binding libraries, applications, and HTTP services—legacy and new—written in a variety of compiled and interpreted languages. In principle "using Java to talk to the network" is not dissimilar from "using XML to talk to the network": either would represent just another layer within a codebase, each with its own advantages and disadvantages.

Finally, note that our aim in comparing JTS with Web Services is didactic, rather than to suggest that our approach to service management should, or even can, supplant the use of Web Services in the VO arena. Our hope is that the more intriguing features of our portal find their way into evolving registry schemes, and that we can find a way to embrace emerging grid and VO standards while not sacrificing the clear gains JTS have delivered in our work. We anticipate that the latter will not be difficult, since the Java platform is by a large margin the dominant vehicle for Web Services deployment (SoapWare.org,[6] InfoWorld survey[7]).

4. Conclusion

Most astronomers remain newcomers to high-performance and grid computing, in part because the barrier to entry remains high: significant cognitive and administrative commitments can be required—at both the institutional and individual scope—to utilize the industrial-strength packages typically employed within these milieux. We have argued that JTS compellingly resolves several aspects of the buy-in problem, and that what one *may* sacrifice by adopting a lighter-weight JTS solution (e.g., peak performance) is adequately compensated for by their ease and autonomy of use, wealth of built-in features, and flexibility. As a partial validation of this assertion we have deployed a space-based VO testbed which augments an existing astronomical toolset, CIAO, with

[5] `http://www.sc2000.org/techpapr/papers/pap.pap261.pdf`

[6] Note the extent (and sometimes exclusivity) of Java support in the listed SOAP implementations.

[7] `http://www.infoworld.com/articles/hn/xml/01/12/21/011221hnjavasurvey.xml`

anonymous discovery of multi-waveband search, retrieval, and analysis capability. Our service coordination infrastructure was implemented in less than 700 lines of Java code, and required virtually zero modification of legacy archive interfaces. The testbed pushes the VO envelope beyond the use of static, faulty, and uncoordinated browser-based services into the realm of fault-tolerant ensembles of dynamically coordinated agents, and to our knowledge represents the first demonstration of an end-to-end analysis thread capable of generating publication-quality results with the ease of use portended for the Virtual Observatory.

While in its present form the testbed should adequately fulfill its purpose as a local proving ground for VO datamodel development, plenty of room for improvement remains, notably in the areas of replication and scalability, interoperability with web-services, adoption of emerging VO standards, streamlining agent interactions with event notification and broader use of JNI, and more graceful co-existence with firewalls.

Acknowledgments. The author would like to thank the entire NVO team at the Harvard-Smithsonian Center for Astrophysics, particularly Stephen Lowe and Michael Harris, for enthusiastic use of, and constructive feedback on, the Tonic and SLgtk packages. This work was supported in part by the Chandra X-Ray Center, under NASA contract NAS8-39073.

References

Batheja, J. & Parashar, M. 2001, Proc. IEEE Cluster Computing, 323

Carriero, N. & Gelernter, D. 1988, ACM SIGPLAN Notices, 23(9), 173

Cresitello-Dittmar, M. et al. 2003, this volume, 65

Davis, D. & Parashar, M. 2002, IEEE Clust. Comp. and the GRID

Freeman, E., Hupfer, S. & Arnold, K. 1999, JavaSpaces: Principles Patterns, and Practice (Reading, MA: Addison-Wesley)

Gelernter, D. 1985, ACM Trans. on Prog. Lang. and Sys. 7(1), 80

Gelernter, D. & Carriero, N. 1992, Comm. of the ACM 35(2), 97

Hawick, K. & James, H. 2001, Proc. IEEE Cluster Computing, 145

Hwang, K. & Xu, Z. 1998, Scalable Parallel Computing (Boston: McGraw-Hill)

Lehman, T. et al. 2001, Computer Networks, 35, 457

Noble, M. & Zlateva, S. 2001, Lecture Notes in Computer Science (Berlin: Springer-Verlag), 2110, 657

Teo, Y. M., Ng, Y. K., & Onggo, B. S. S. 2002, Proc. 16th IEEE Work. on Parallel and Distrib. Simulation, 3

Wyckoff, P. et al. 1998, IBM Systems Journal, 37(3), 454