Although every aspect of autonomic computing is a significant research challenge [2], it is necessary to commence designing and developing tools and methods towards the goal of autonomic computing [3]. While different aspects of autonomic computing have been explored in isolation, there is a lack of effective autonomic infrastructure that enables programmers to develop and integrate their programs using autonomic primitives. Although there is general agreement on the structure of the autonomic element, there are no general implementation details of the internal architecture of the autonomic component [3]. The lack of a common implementation of this basic autonomic component hinders the interactions among different self-managed technologies. Since most other research employs proprietary techniques to develop such components, it is difficult for programmers to incorporate autonomic properties in the design and development of such systems. The programmer is left still more concerned with learning and implementing such proprietary techniques rather than concentrating on the actual problem in hand.

This paper presents an architectural design of the standard autonomic element and simulation in a realistic environment. Having an open and flexible structure for the most basic component of any autonomic system alleviates programmers from the complexities associated with designing, developing, integrating and managing autonomic systems. Design goals of the presented structure of the autonomic element are as follows:

- It should be transparent to use and programmers should “program as usual” with minimal constraints.
- It should employ open standards and common metaphors to develop the autonomic element.
- It should be lightweight and should not consume system resources unnecessarily.

This paper presents a Java-based autonomic element design that greatly simplifies prototyping of self-management techniques. Although this design is based on Java, any programming language that supports standard object oriented metaphors can be easily used to develop the equivalent model of the autonomic element. The design allows different techniques to be compared and helps investigate the interaction among different techniques. The implementation of the architecture is incorporated into a Java-based distributed autonomic infrastructure [4] to assist the rapid deployment of user applications and easy-to-use transparent, self-management of distributed systems.
The architecture presented in this paper supports computational and data intensive centralized application where the computation-to-communication ratio is significant. The proposed approach models a centralized application in terms of a graph consisting of application components and communications among them. The centralized application is first statically analyzed to derive the application components and their interactions. A partitioning scheme is then utilized to generate several partitions (consists of a single components or grasps of components) so that highly communicating components are placed in the same partition and the overall communication cost is minimized. In this architecture, these partitions are actually treated as managed elements inside the autonomic elements. Readers interested in the detail description of static analysis and related transformations are referred to [5, 6].

2. ARCHITECTURE

Autonomic elements are the heart of any autonomic system. An autonomic element is described by two distinct parts. The autonomic manager provides the functional abilities of the element and the managed element is the entity that the autonomic manager is monitoring and controlling. All autonomic elements have a control loop (MAPE-Monitor, Analyze, Plan, Execute) that dictates the work flow of the different sub-components of the autonomic element. Figure 1 shows the design for the autonomic element.

An autonomic element is envisioned as a multi-threaded server (daemon) having multiple components (monitor, analyze, plan and execute) and interfaces (sensors and effectors) running concurrently for better CPU utilization. Another alternative is to design the sub-components as threaded objects and schedule them according to a control loop. As Java does not support language level thread control (stop, suspend etc. are deprecated as of Java 1.2), it is not possible to write a scheduler for Java threads without sacrificing performance. Writing such a scheduler is a complicated programming task and makes the autonomic element bulky and consumes more CPU power for scheduling. Instead, this paper proposes each sub-component as a threaded object and delegates the scheduling decisions to the underlying JVM. The benefit of having a threaded model for the MAPE control loop over a sequential loop is:

- Better CPU utilization by concurrently executing the sub-components. Since multi-core and multi-processor machines are more affordable and common then ever a concurrent MAPE loop allows autonomic elements to respond quickly to environment changes.
- Algorithms which are responsible for different aspects of self-management can easily be incorporated as a child thread inside any of these parent threads denoting the four major components of the autonomic element.
- Better throughput for information processing as no sub-component will be blocked waiting for another slower processing sub-component. Having a sequential control loop can block the execution of the whole element when one component is blocked for any reason and therefore may have an adverse performance effect.

To have the same semantic functionality as the control loop, this design uses semaphores in strategic places to order the execution of the sub components and synchronize the threads when accessing shared data. This way, the control loop in this design is non-blocking due to concurrent execution of the individual sub-components. This allows faster execution of autonomic behaviors but introduces the following challenges:

- Asynchronous execution: Since each component is running asynchronously, it has to be guaranteed that no internal messages are passed inadvertently to a different component.
- Deadlock and starvation: Care must be taken during development of shared data structures and data components such that access to those shared resources are mutually exclusive.

There are separate threads for environment interfaces (sensors and effectors) and for the main components (monitor, analyze, etc.) of the autonomic element. From

![Figure 1. Internal Architecture of the Autonomic Element.](image-url)
time to time, the autonomic element has to perform its own management and accounting tasks. So the design is extended beyond the standard notion of autonomic elements with an element manager and several control interfaces. As shown in Figure 2, the autonomic manager assumes the role (setup or active, see Section 3) that is being required of it. Some of the autonomic elements in the system are allocated some higher level administrative authority. These managerial autonomic elements will either manage system registry and policy depository or will act as the user interface for program partitioning and transformation, monitoring or the source or destination of program input and output. However, all the autonomic elements in the system have the same properties and they could act in any of the above roles if they are instructed to by other autonomic element.

There are multiple interfaces for the different services to be described, discovered and supplied. For instance, the Service interface allows other autonomic elements to reach a service agreement with an autonomic element. The Policy interface provides a way to transfer and modify policies between different autonomic elements. The Monitoring interface provides methods to monitor each autonomic element’s internal activities and status information. The Deployment interface provides methods through which managed elements are allocated, deployed or restarted. Separating the functional aspects of autonomic management from the management of the autonomic element itself makes the overall software architecture more modular and extensible.

3. LIFE CYCLE

The life cycle for the autonomic element is shown in Figure 3. Once the autonomic element is initiated, it moves into the setup state. In this state, there is no managed element attached to the autonomic element and the autonomic element is acting basically as a bootstrap manager for future deployment of the autonomic element with a managed element. In this state, the element remains in the hibernate state until either:

a. The autonomic element is upgraded with a new version: Each of the sub-components of the autonomic element is implemented using a technique described in [7]. Explicitly, the functional part of a sub-component is implemented through a common interface so that at runtime it can be updated, if necessary, to fulfill user policy and goals. The functional code resides inside a class (task.java) which implements a common interface named (taskInterface.java). At runtime, the old task class is replaced by any newer version of its implementation. This allows adaptive composition of the component by adding, deleting or modifying the algorithm of each sub-component during run-time. This provides a general framework for the activities needed for the self-management of systems without forcing a specific programming model for the implementation of the actual operations. For instance, the programmer now has the freedom to implement the plan algorithm as either an artificial intelligence planner or a graph-based solution leveraging domain-specific knowledge and can switch between multiple algorithms during run-time without jeopardizing program execution.

b. Autonomic management of a managed element is sought and the autonomic element switches to the active state: If such a request is received, the element manager first creates all the necessary threads for the autonomic element (install) to deploy. The threads are then configured with any initialization values (configure) and their execution is initiated (deploy).
and the state of the autonomic element is moved to the active state. In the active state, all threads are subject to the underlying JVM thread model and can have multiple internal states as the platform dependent implementation of the threads. If the managed element is no longer to be managed by the autonomic element, it is switched back to the setup state and remains in the hibernate position until a new phase begins.

c. The autonomic manager receives a stop signal from the administrator to stop execution.

The setup state executes in a sequential fashion to guarantee that the sub-states follow a pre-determined path of execution. Whereas the active state follows a parallel execution, where each of the sub-states follows its own execution path determined by the underlying thread scheduler and communicates with each other via synchronized semaphores.

4. IMPLEMENTATION DETAILS

4.1 Data Structures

The choice of data structures has a profound impact on the performance of the element. Use of efficient data structures ensures small memory usage and faster execution of the element as a whole.

4.1.1 Data Atoms. All internal communication is performed in a standard format, named atoms and molecules. Atoms normally travel between different components in the direction of the control flow. An atom is essentially a collection of XML tokens which can be interpreted by all the sub-components of the autonomic element. The main reason for selecting XML to represent internal data is that it gives flexibility for future improvement and extension. Since XML is machine agnostic, autonomic elements developed in other programming languages can be incorporated easily into an existing system. Data atoms have the capability of chaining $n$ atoms together to form a single molecule. Atoms need to chain to each other when the occurrence of certain events requires a steady supply of atoms which have some form of common relationship. However, care needs to be taken with respect to the size of a molecule (upper bound for $n$), as making an excessively sized data molecule can slow down the autonomic element. The self-optimization aspect of the autonomic element can dynamically change the size of a molecule (value for $n$) by monitoring the element over long runs.

4.1.2 Priority Queue. Each of the components inside the autonomic element has a buffer implemented as a priority queue. As data molecules are flowing through the control loop, they are stored temporarily inside the components for processing. Different components can create new data molecules after processing existing data atoms and can pass those to the next component. This design permits at most one molecule to pass between components at any given time. This ensures a consistent flow of information among the components and ensures that individual components are not overwhelmed. Therefore, molecules are sorted and stored in the queue based on their priority and are processed according to the highest priority. Normally the size of the queue starts with a predefined size. However, over time it may grow or shrink depending on the flow of molecules in the system.

4.1.3 Locker. This is a special data structure that resembles an operating system semaphore with added functionality. Lockers are placed between each pair of sub-components in the autonomic element to transfer atoms between them. The main responsibility of the locker is to receive an atom when it is empty and notify the destination component of an incoming atom. Although it is desirable to keep the communication mechanism open between the individual sub-components from an architectural standpoint, lockers are introduced to pass information (data molecules) between components for two main reasons. Firstly, to avoid any deadlock situation between two concurrent components during data transfer. Secondly, to avoid wasting any processing time of the components unnecessarily. Instead of the components polling each other for molecules, the locker notifies the corresponding component if there is a new molecule awaiting processing. Polling introduces synchronization and deadlock issues and a component is expected to spend most of its allotted schedule time polling for incoming atoms. Having all atoms transfer through the locker improves the response time of the components and does not block the receiver if the sender is processing at a slower rate than itself for any particular reason. All lockers fulfill following two conditions:

- The locker is always unidirectional. Therefore, only one component can push a molecule in a locker and only the adjacent component can pop it. To loop an atom back to a previous component, it has to passed around the control loop and appear in the queue of the destination component. It is possible to have a bi-directional locker between components; however, care should be taken to avoid any possibility of deadlock. A bi-directional locker does not increase performance, rather complicates the whole design. In the current version of the design bi-directional lockers are not permitted, but with extended behavior modeling and verification, this could possibly be implemented; but no significant gain is expected from doing so.

- As with semaphores, the push and pop operations must be atomic. Since Java does not provide any language level abstraction for atomic methods, we used the atomic reference object implementation in java.util.concurrent.atomicReference of the Java 1.5 API to implement those methods. So the methods have an atomic object reference, where the data atom can be pushed or popped atomically. The use of this class of variable offers higher performance than is available by using standard synchronization techniques.
4.1.4 Data Streams. These are one-way dedicated communication channels between two entities. Only one thread can write into it and another can read from it. To avoid any deadlock situation, all read and write operations are forced to be atomic as described earlier. Data streams are used where no intermediate processing is required, such as in the sensors and effectors to provide a rapid response.

4.1.5 Hooks. This is a collection of data structures and methods that allow the sensors and effectors to actually interact with the environment or the managed element. The representation for such hooks may not be universal because of the diversity of applications and environments. The development of such hooks is domain specific and should be addressed on a per system/application basis. For the application domain utilized in this paper [4], byte code segments are injected at strategic places in the distributed Java objects which are treated as the managed element. This sort of code injection [5] is completely transparent to the programmer and performed during preprocessing and runtime and before deployment with an autonomic element.

4.1.6 Atom Repository. Any processed molecules that the system wants to store for future use are stored in the atom repository. This is identical to the knowledge part of the autonomic element. Along with storing atoms, it can also store rules and policies regarding different functional aspects of the autonomic element. Finding a good knowledge representation is a separate research challenge. This design used ACPL [8] to represent rules and policies as it provides a user friendly form of policy definition, policy management and appropriate tools through an API to work with policies. Since ACPL is based on XML, this permits atoms to be seamlessly incorporated into the repository. Although multiple components can read simultaneously from the repository, only one component can write to it at any time. This is to ensure consistency of the data in the repository.

4.2 Operation of the Sub-components

4.2.1 Sensors and Effectors. There are two sensors in every autonomic element. One is responsible for interfacing with the environment and other is necessary to interface with the managed element. The common responsibility is to acquire runtime information and incoming messages from the managed element or environment respectively. The sensors are not heavy weight processes as they only check the validity and format of any incoming message and forward them to the monitor component. On the other hand, effectors receive data from the execute component and verify its format and then either forward it to the environment or invoke appropriate methods inside the managed element to effect its execution.

4.2.2 Sub-components. Usually each component takes molecules from its input source (locker or sensors) and processes it and puts it in the priority queue. At each scheduling cycle, it checks whether there is anything in the queue which needs further processing or needs to be passed to the next stage of execution. If the corresponding locker or effector is free, it then pushes the top most data atom to the corresponding destination. Data molecules may not propagate all the way to the last component (execute) if it is internal to the autonomic element or there is no further processing available or necessary for a particular data molecule. Therefore the execute sub-component has data sink point, which is basically an object nullifier. The garbage collector is called to free up the memory of those nullified objects. Since the Java garbage collector blocks all threads during garbage collection, calling it frequently decreases the performance of the autonomic element. So, different heuristics (such as waiting until the number of nullified objects reaches a minimum threshold or the queue size reaches a specific length) are used to decide when to call the garbage collector.

5. PERFORMANCE ANALYSIS

A test bed environment is developed to simulate the behavior of the proposed autonomic element in a real environment with different parameters, such as input rate and the size of atom. This allows observation of how components interact with each other to be made and how much time they really spend for their own management to be measured. In executing the autonomic element, the following techniques are used to optimize the operation:

a. Classes for the components are designed in a modular fashion and class loading is performed only after determining that a particular module is needed. This provides a fast startup and keeps a small memory footprint when the element is not used. The drawback is that, response time may increase as classes have to load dynamically at runtime. However with proper class caching techniques and designing effective class loaders, this effect can be minimized.

b. Management of atoms (sorting, duplicate matching, updating, deleting etc.) in the repository is only performed when the element is idle.

All experiments were performed on an eight processor (900 MHz each) SPARC Sun server running Solaris 9. A sequential version of the autonomic element is also implemented to compare the concurrent version against. In both cases, the MAPE loop has the same amount of processing on the intermediate atoms. Running both versions of the autonomic element in a single processor machine provides the sequential version a significant performance boost as there is no overhead related to thread scheduling. Since machines with multiple processors or cores are becoming increasingly common now-a-days, the concurrent version can exploit this architecture. Therefore, from our viewpoint, it is sensible to measure throughput...
over a fixed amount of time in both versions instead of measuring execution time of an atom or a number of atoms. To measure throughput, both versions are run for a pre-set amount of time and are both flooded with data molecules at the same input rate. It is measured how many atoms both versions of the control loop can handle within that set amount of time. Figure 4 shows the number of atoms that both the versions can handle for different data molecule sizes. The y-axis in Figure 4 is represented using a logarithmic scale to better signify the difference for smaller values. It is evident that the proposed concurrent version has a higher throughput than the sequential version, especially when the size of the data molecule increases substantially. On average, the concurrent version processed 57% more information than the sequential version in a fixed amount of time.

To observe how the concurrent architecture behaves in the case of different atom sizes and input rate, the concurrent version is executed with different atom sizes and different input rates for a fixed period of time. Figure 5 shows the throughput of the concurrent model in the case of various input rates and atom sizes. From the figure, it is obvious that the model is performing as expected by handling more atoms when the atom size is small and the number of atoms entering the system is high. The throughput decreases as the atom size increases and the input rate decreases. With 16K atom size, the throughput becomes independent of the input rate as our fixed period is insufficient to handle such large atoms.

Table 1 shows different statistics related to the two versions of the control loop. In the concurrent version, the number of source code files (required to implement the architecture) may reduce if dynamic composition is not required, however in real-life situations, it is inevitable and we have to be comfortable with the added code inflation and class loading time. Both versions of the control loop are executed with the same input rate, atom size and for the same amount of time and are profiled using the NetBeans profiler [9]. Since the sequential version has a smaller number of runtime threads than the concurrent version, the maximum amount of heap used during the runtime is lower than the maximum heap used by the concurrent version. As a result, the sequential version was spends 4.8% time less doing Garbage Collection (GC). Although this profile can change for different runs and for different parameters, the added cost for the concurrent version is worth taking for improved throughput, dynamic composition and better response time.

6. RELATED WORK

There have been several autonomic system projects that describe autonomic infrastructure and define the architectural aspects of an autonomic element. However these research projects address the issue of designing an autonomic element with different objectives than ours. The goal of this research is to make the autonomic elements simple to program and use and which in turn is universal enough that it can be used in most cases. The work by Jarrett et al. [10] has the same objective as this paper. The authors describe an autonomic computing architecture and accompanying implementation infrastructure on top of the Cognitive Agent Architecture. However, they did not provide an architectural implementation of the autonomic element itself.

Wang et al. [11] proposes an autonomic element design based on the mind agent model. It is a promising approach to meet some challenges in autonomic computing initiative; however the authors actually did not implement the design to observe its workings in a real environment.

Accord [12] provides some good ideas on how to design autonomic elements and identifies the issues that should be addressed. However, they did not provide any architectural design. Similar to this work, AutoMate [13] presents a prototype autonomic system without defining the internal workings of the autonomic elements.

Autonomia [14] proposed design of autonomic elements using proprietary techniques that add additional complexity and a steep learning curve for the programmer.

IBM Autonomic Toolkit [15] provides tools and an API for monitoring, analysis, planning, and executing autonomic applications. Although these tools provide some
inter-dependent functionality for an autonomic environment, they however, do not address the development of autonomic elements.

Existing research on multi-agent systems is a rich source of good ideas about system-level architectures and software engineering practices [16, 17] for autonomic computing. System such as Unity [18] uses multi-agent paradigms for developing autonomic systems. However, these approaches lack common agreement on the exact design of most agents. The autonomic computing community could certainly benefit from their experiences.

7. CONCLUSIONS AND FUTURE WORK

An autonomic element is the fundamental building block of any autonomic application and system. Although different aspects of autonomic computing are researched in isolation, the structural operation of an autonomic element has not been fully modeled. The standard definition for an autonomic element does not give a clear picture of how to build one from scratch and several proprietary designs have been proposed that are not interoperable with each other. This paper presents an engineering perspective of building a domain independent autonomic element. It is important to have a well defined model of the basic building block to develop autonomic systems. The architectural design presented is self-regulating (in the sense that multiple internal sub-components are running in parallel but the internal data flow is consistent and unidirectional) and uses standard object oriented primitives and software engineering techniques to make it easy to develop and implement.

Analysis of runtime behavior and profiling shows that the design is sound and can work smoothly without causing any deadlock or producing extensive overhead. Future works include the implementation of the same design with other programming languages and incorporation in a system where autonomic elements from different programming language are in play.

8. REFERENCE