State Considerations in Distributed Systems

by Daniel W. Goldberg

Introduction

Maintaining state in a distributed system is a critical task that has received much attention in operating systems literature. The "state" that a distributed system maintains is dependent on the goals and services provided by the system and, as such, can take a variety of forms and serve a host of different purposes. Although many paradigms addressing this problem of state maintenance in distributed systems have been presented, this research paper will discuss two in particular: leases, as first introduced in [5], and soft state, as first introduced in [2] and further refined in [10].

Even though soft state and leases are closely related, an interesting difference between them arises when one considers the level of assurance that each participating party has regarding their knowledge/belief of the shared state in the system. With leases, each party can rely on the fact that the other is obligated to the contract they have made with each other. In a sense, this state model can be described as "discrete" because it is well-defined. This is opposed to the "non-discrete" case in soft state, where there are no such guarantees or contracts, and the relationships between parties are fuzzy and not well-defined. The differences between these two paradigms affect the assumptions that the parties can make about the system and also dictate different styles of recovery, consistency, and operational overhead, which will be explored in this research paper.

The analysis presented in this work will be cast in the light of two of the most well known distributed systems: Jini [13, 14], which uses leases, and Scalable Network Services [4] (SNS), which uses soft state. A background discussion will be provided on the use of state in each system, highlighting what information is maintained for what purposes. The work will next explore the reasons that particular models were chosen in each case through a focused discussion comparing and contrasting the design goals of each system and how each state model helps achieve them. Subsequently, an assessment will be presented describing the possible effects on each system, had the other state model been chosen. Throughout the paper, the discussion will focus on how each state model affects the level of consistency afforded, recovery procedures, and amount of overhead required to facilitate each.

Soft State and Scalable Network Services

In his review of the early work leading to the development of the modern Internet [2], David Clark briefly identified a sub-optimality in the design of the DARPA Internet Protocols, in that resource management decisions could not be made at the gateway level because the gateways only understood packets, not the streams they were composed of. He hypothesizes about a state management scheme called "soft state," in which endpoints in a network "flow" inform the gateways in between about the flow periodically, so that better decisions can be made. By broadcasting this information periodically, the system would be able to handle failures more efficiently, while benefiting from the higher level state. This first introduction of soft state did not fully illustrate the power that such a state management scheme could provide, and the definition was further refined and formalized by [10].

SNS, as described by [4], offers a new framework to use and exploit clusters of cheap, commodity machines to achieve high performance, scalability, and cost-effectiveness. In their work, they describe how the traditional problems associated with running and maintaining large clusters (administration, component replication, partial failures, and shared state) can be overcome if one relaxes the ACID (atomic, consistent, isolated, and durable) constraints that are placed on a system. They offer a new paradigm, BASE (basically available, soft state, and eventual consistency), which they argue can effectively be used to afford an acceptable quality of service to the users, while simplifying the implementation and administration of a large cluster of commodity machines. The key to their approach is the development of a reusable SNS layer that separates the content of the service provided by the system from the implementation that provides the service. They demonstrate how this layer can be effectively applied as a "black box" component to two problem instances: 1) the Inktomi Search Engine and 2) a JPEG distiller engine [4]. The state that is maintained in this system is the availability of processes (workers and manager) and the load of worker processes. Of these, the first is maintained by soft state, and the second is derived from the manager's distribution of work to the workers. The availability of processes (workers) is used to determine when a process has crashed and when to initiate recovery. The availability of the manager is used to determine when a worker should promote itself to manager, in the case that the manager has died. Together, these two operations ensure a robust system in the face of inevitable system failures [4].

Central to this SNS layer is the use of soft state to achieve the goals of high performance and failure management [4]. This model was chosen over a traditional "hard state" approach, where the same state must be maintained between peers, for two reasons. First, in their implementation (and recognized by others [10]), soft state enables fault tolerance to become trivial because crash recovery becomes implicit [4]. Each peer monitors the others by the use of "process peer fault tolerance" and reboots it if and when it fails. Upon restarting, the failed peer simply listens to the beacon messages being passed and incrementally regains the state of the system [4]. Second, soft state allows high performance because it enables the effortless incremental addition of new

"workers" when the system becomes overloaded. The state beacons sent throughout the system as multicast are picked up by the new workers as they boot and allow them to quickly come online and into production [4], much like the gateways described by [2]. This is opposed to hard state, where state must be explicitly installed and removed per each worker as unicast, which suffers from the fact that as the network grows, the number of messages will increase and are not scalable [12].

However, the use of soft state is not without its drawbacks. By relaxing the ACID constraints to BASE, the consistency throughout the system suffers. Using soft state, the level of consistency, a quantification or measure of how closely different objects in the system perceive the same value for the same entity [6], is determined by the frequency of the beacon messages. This is an operational overhead that is not present in hard state systems because the state is explicitly installed and persists until is it explicitly removed. In choosing the correct frequency, one must consider the trade-off between letting the state get stale by sending messages infrequently and the amount of processing required to update the state as each new state message arrives. This phenomenon has been observed and studied extensively by [6, 8, 12] to determine optimal methods for assigning the state message frequencies, but it was not explicitly addressed in [4]. However, because "eventual consistency" is a primary goal of BASE, the authors of SNS realized that by using soft state, they would be sacrificing consistency as a whole at the benefit of easy scalability, higher performance, and quick recovery [4]. For the particular applications that SNS was developed for, this was not considered to be a problem because they perceived that "an approximate answer delivered quickly is more useful than the exact answer delivered slowly" [4]. Thus, even though not explicitly stated, they were willing to accept the overhead of redundant periodic rebroadcasting of state, even though the state may not have changed, which was identified as a weakness of soft state by [10]. Additionally, the soft state assumes that if failures do occur, they are independent of each other and would not represent a whole site going down at once [7].

Leasing and Jini

Leases were first described by [5] as a method of ensuring consistency in distributed file caches. In this scheme, a client specifically requests a service or object from a server for a period of time, to which the server may accept or reject. If accepted, the server grants the lessee specific rights to the service/object in question for a specific period of time. The server ensures that either the lessee will be the only party to have access to the service/object (if it is an exclusive lease) or that the server will inform the lessee of any changes made to the service/object during the lease period (if it is a non-exclusive lease). After the expiration of the lease, the lessee no longer has any guarantees about the service/object, but the lessee can renew the lease prior to its expiration to maintain exclusive control [5].

While this first description outlines the basic paradigm of the leasing mechanism, it is focused on the distributed cache consistency problem alone. The Jini architecture, defined as part of the Sun Microsystems Java application environment, uses leases to maintain the state of a flexible distributed service environment based on federated groups of users and services. Jini is separated into three components: 1) the infrastructure that provides the building blocks for composing a federated system, 2) the programming model that enables services to communicate with each other, and 3) the services that are offered in the networks themselves. Within the infrastructure resides the lookup service that allows clients to find services, and the lookup service allows services to publish their availability based on the join and discovery protocols. The programming model defines the interfaces that are used to lease services and be notified of changes [13, 14].

In the Jini architecture, the primary goal is to achieve flexibility in terms of easy addition/removal/updating of services, leading ultimately to scalability and fault tolerance because more services can be added easily, and redundancy can be increased. The primary method used to achieve this is by maintaining the state of the system at the lookup server and enforcing the use of leases on a per-service basis. In particular, the state of the system consists of the set of services that are presently offered at any point in time. This global state of the system is achieved by the services registering themselves through the join protocol, where they inform the lookup server, or multiple lookup servers, of their existence and what services they offer (more specifically, what interfaces they implement). To enable flexibility of services coming and going at will, the client and server negotiate a lease contract specifying the rights to the service for a period of time. For the time period specified in the lease, the client knows exactly what the service will be, but after it expires, all bets are off, and the server is free to change the service or leave the system. This combination of the lookup server for service discovery and time-based access to services using leases facilitates the goals of the system [13, 14].

Recovery in the Jini system is fundamentally different than that of the SNS system. In SNS, they assumed that machines would be going down all the time *because they crashed* and that recovery of the *same service* would be critical. In Jini, this assumption is not made. Rather, they assume that machines (actually the services) will *intentionally be leaving* the network *without crashing* and would otherwise be available. The leasing framework provides for this explicitly through the use of the time-based contract. The user of the Jini service knows exactly the timeframe that they can use the service. In SNS, a crash can happen at any time, unexpectedly, without warning.

It becomes obvious that a significant amount of overhead occurs during the discovery and join protocols as well as during the negotiation and renewal of the lease. During the publication of a service, a service must contact several lookup servers and send the same information to each as multicast, describing itself so that it may be found, with responses from each coming back as unicast. During the discovery of a service, a client may potentially have to contact several lookup servers to find the appropriate interface implementation that it is looking for as multicast, and a client, again, may get multiple responses as unicast. While this "rendezvous through a third party" [1] imposes a layer that the two parties (server and client) must initially meet through, the use of multicast in one direction at least makes it more efficient than a simple broadcast strategy.

Much like soft state, the issue of message frequency becomes an issue, and it introduces additional overhead into the system due to the state model. In both cases, if the state has not changed (in soft state), or if no one else is contending for the lease, then the messages were wasted because they are not contributing to any new state in the system [5]. Additionally, the length of the lease chosen will affect the performance of the system, creating a "trade-off between minimizing lease extension overhead versus false sharing" [5], where false sharing is defined as "a lease conflict when no actual conflict exists"

[5]. In other words, when one party is holding the lease and not using it, another could have been using the leased object/service but was denied because the lease was already out. Again, like soft state, this lease term selection is an active area of research, and optimal strategies for deciding it have been studied in [3, 9, 11].

Perhaps the most stringent assumption made of a lease-based system is that the clocks of the lessee and landlord need to be synchronized. If the server's clock is fast, it may allow invalid access to leased services/objects before a lease is up. If a client's clock is slow, it may be using an expired lease without knowing it [5]. Therefore, the additional overhead of maintaining clock synchronization is introduced into systems using leases.

SNS vs. Jini: Design Goals and State Choices

A fundamental difference exists between the services offered by SNS and Jini. SNS is used to provide a scalable, fault tolerant system for a *single service*, and Jini provides a scalable, fault tolerant system for *multiple services*. With this in mind, one can see that the choice of state models used by each system was not made arbitrarily. SNS benefits from using soft state because it allows the very limited amount of state that the system maintains (what processes are alive and workloads) to be passed around efficiently between processes. Taking into account the realities of cluster computing that the designers faced, that machines were necessarily unstable and that ease of maintenance was preferable to tight consistency, the soft state paradigm was their best option because it offered an implicit mechanism to manage failures and add nodes quickly as load increased [4]. With all machines in the cluster working on parallelizing and scaling the *same service*, the soft state model simplifies the storage and dissemination of the shared state throughout the process as a whole (composed of each of the components workers). This is necessary for the system to achieve the design goals of being easy to manage and robust as a whole [4].

Jini, on the other hand, chose their leasing state model primarily because of the explicit flexibility that it provides [13, 14]. As mentioned, the designers sought a system that allowed for the easy inclusion and exit of services. The leasing model imposes the constraint on the participants that they can only trust a foreign party for a finite time frame, after which they must reacquire the contract. By bounding the time that any service is required to be in the system, as per a lessee counting on the service to be provided, the Jini architecture allows for services to come and go as they wish, when they need to. A service simply needs to stop accepting leases and wait for all existing ones to expire if it wishes to leave the system in a "valid" state. This is opposed to an unexpected crashing if the service leaves the system when clients are still depending on it, putting the system into an "invalid" state. Furthermore, the use of leases achieves the goals of the Jini system in that it allows each service to define its own state during the contract negotiation with the prospective client, and it does not require all clients to maintain the same state. This flexibility ensures that *many services* can be offered *at the same time* within the architecture, each with their own particular state or constraints specific to the needs of the client and the capabilities of the server.

Soft State vs. Leasing

With a clearer picture of the differences between the soft state and leasing paradigms, we can now draw some distinctions between the two. If we take a step back and look at the assumptions that the components can make of the system as a whole when using soft state, we notice some interesting phenomena. First, there are no explicit contracts associated between parties. As such, no component of the system can have any significant level of trust in the availability or quality of services provided by others, creating "fuzzy" relationships between parties. This relationship can be described as "non-discrete," in that the boundaries of trust and reliability between parties are not well-defined. The parties cannot base their operational decisions on guarantees in terms of reliability or trust made by other components because there are no such guarantees, and no assumptions can be made. Therefore, the system functions as a set of semi-interoperating, loosely-coupled, *independent* entities collaborating to achieve a common goal or work on a common task.

As shown in SNS, this approach can successfully be employed in highly parallelizable situations where each component of the system performs the same task, such as Web request processing. In these types of systems, soft state excels because it allows the system to handle failures and scale well, while providing an "acceptable" level of service [4]. The level of service is merely "acceptable," meaning that it is not the optimal, but it is good enough for the task at hand. The periodic messages that enable the robustness of the system as a whole also limit the quality of the results returned because the components may be working with old state information. In SNS in particular, the manager may not know that some sets of workers have been rebooted and are ready for action again until their next state broadcast. Therefore, it will not assign them work, even though they are ready and other workers may be overloaded. This is the crux of the "eventual consistency" in the BASE paradigm. Although fine in the SNS context, it should be noted that this approach would not work well in systems that needed explicit guarantees of service availability and quality, such as real-time operating systems.

In contrast to the non-discrete nature of soft state, leasing can be characterized as "discrete" because it dictates a formal arrangement between the parties who are interacting. The service grants a lease to the client for a specific period of time with specific privileges [5]. By doing this action, each party knows exactly what to expect from the other, and they can base their decisions and assumptions on the terms agreed to in the contract. During the lease time period, the client can assume that the service will honor its agreement in providing the services it asked for, and the server knows in advance explicitly what its workload will be, and it can make leasing decisions to achieve and optimize its own performance goals. In this case, the boundaries of trust and reliability between the parties are clear and well-defined.

As shown in the Jini architecture, this approach can successfully be used to offer a clear set of well-defined, yet flexible services in a distributed networked environment [13, 14]. In the type of system that Jini was striving for (an easily extensible and flexible one), this approach is ideal. As previously mentioned, the discreteness of the contract between the parties enables a level of assuredness during interactions, yet it provides a flexible mechanism by which parties can come and go in the system at will. The Jini design goal of a "federated system" implies that the local processes have control, and the centralized services merely facilitate this [14]. This constraint is fulfilled by allowing the negotiations of service agreements to be performed at the peer level through the use of leases, not in a centralized

server. However, it should be noted that the Jini architecture is not designed for systems that fail intermittently and frequently. The recovery procedures for systems with such constraints (like clusters of commodity machines) are not well-served by the leasing mechanism because of the negotiations that must occur before services can be rendered between clients and servers.

Scalable Network Services with Leases

While difficult to imagine, it is interesting to explore what would happen to SNS if it was implemented using a lease state management protocol instead of soft state. In one possible scenario, the workers would negotiate a lease with the manager. The information negotiated would include the time period of the lease and the amount of traffic that the workers desired during this period. Then, for the period of the lease, the worker could make assumptions about the amount of traffic it will be receiving, and the manager could make assumptions about the overall distribution of work in the system because it will be explicitly maintaining the number of available workers with the amount of work going to each. The benefit to this arrangement is that the state maintained in the system has tighter consistency because all workers must obtain leases to get work. Therefore, the manager has a complete and accurate picture of the entire workload of the system and will be able to better direct load to free resources and anticipate the need to spawn new workers based on present system capacity.

However, this arrangement as it has been explained so far does not take into account the fact that the system is operating on a cluster of not-so-reliable machines that can potentially crash at any moment. If a machine was to crash after having work assigned to it, there would be no way for the manager to be informed, and it would keep sending traffic to it. All of this work would be wasted because it would not be answered. The lease will eventually time out, and the traffic will stop being sent, but there still has been no communication to tell any other component of the system that one has crashed and needs to be rebooted. To address the first problem of wasted work being sent to dead machines, the manager could impose a short lease time. Increasing the overhead required in renewing leases for machines that are operational would limit the amount of potentially lost (unprocessed) traffic sent to dead hosts.

To address the second problem of reviving dead processes, another scheme could be devised in which workers are required to create a lease with each other that maintains the state of their liveliness as well as a lease with the manager that reports the liveliness of other workers. In this case, the workers could include a service that requires worker peers to perform a function signaling their liveliness on each, while necessitating that the between-worker lease be continually renewed. If not renewed, the state of the worker reverts to the default state, implying that the other worker is dead and needs to be rebooted. In other words, this would be a, "Hello. I'm still alive; please don't reboot me" service that the workers would call on each other.

The next lease between the manager and the workers would serve as the ultimate reboot caller for the dead workers. This is required in the case that multiple workers who are monitoring each other go dead simultaneously. If only the workers were monitoring each other in this situation, neither of the dead workers would ever be rebooted because neither would be alive to signal each other to come back to life. To avoid this, the workers could be required to take out a second lease with the manager, or, perhaps, they could modify the first one to include a list of workers for whom they are the "watchdog." The server would then know who had died when leases stop being renewed for a particular host or if a watchdog reports that the ones it was watching are no longer alive. Similarly, a leasing scheme would have to be designed to promote a worker to manager status in the case of the manager dying.

It should be clear at this point that implementing SNS with leases creates the exact problems that [4] were trying to avoid. Using leases introduces the need for complicated logic into the system to detect and recover from failures. Even though the leasing strategy proposed in this section may not be the optimal one, it illustrates how one would need to pick apart the requirements of the system one-by-one and determine an appropriate leasing scheme to address each. In particular, this shows how the benefits of using a cluster start to become outweighed by the management complexity required to keep it running. The beauty of SNS with soft state is that it offers a simple, cost-effective method for automatically and implicitly managing recovery in a fault-likely environment.

Jini with Soft State

Equally interesting is an exploration into the ramifications of implementing the Jini architecture with soft state instead of leases. In keeping with the Jini architectural goal of providing a flexible service, one could imagine this scenario unfolding in a situation where the services offered by hosts were periodically transmitted directly out onto the network. This information could be picked up by the lookup servers, which could then transmit the total state of the system to the awaiting clients. However, this step would become redundant as the clients could pick up the state broadcasts from the service servers just as easily. This state information would include server availability and the services provided. In this case, where the need for the lookup servers has been removed, the system would become more decentralized, with each client knowing the entire state of the whole system, i.e., where services are located.

This may seem at first to be a good idea, but when one considers that another primary goal of the system is to enable scalability, this idea of each service provider continually publishing its state becomes unrealistic. As the number of hosts increases, so will the amount of network traffic. By increasing the number of hosts high enough, the majority of the traffic on the network will become state messages, thus reducing the amount of bandwidth that can be used for the actual services. This problem is particularly bad in the Jini model because of its desire to allow a rich set of diverse services. Each of these services is distinct and needs to publish its own unique state so that their particular service can be discovered by a client wishing to utilize them. This prohibits any type of aggregation of state messages that could be performed, such as in the case where all of the servers were offering the same service—for example, in SNS, where choosing any worker will do because they are all doing the same job.

As previously mentioned, this first problem of state message overhead can be overcome by making a trade-off in the "softness" of the state maintained by decreasing the frequency of state messages, at the price of reducing the consistency of the system (see [6, 8, 12]). However, the original Jini model ensured very tight consistency because once a client and server negotiated a contract for service, the client was able to make explicit assumptions about the availability and quality of that service. If one were to try to reduce the messaging overhead by decreasing

the frequency, it would mean that the information clients had about the state of the services would be outdated and possibly no longer valid, thus raising the level of inconsistency in the system as a whole.

One could image using the advanced state refresh timing messages detailed in [6, 8, 12] to pick better refresh intervals, but this would simply serve to vary the level of consistency offered *per each service*, because each service would be optimized independently. This, perhaps, is even worse than having a single looser value for consistency because, at least with that, a client would be able to bound just how bad their information would be for any particular combination of services, instead of having to maintain this on a per-service basis and calculate a total value for the whole system.

The overall effect of implementing the Jini architecture using soft state is that the foremost principle of the original architecture, that services can count on the contract negotiated to be valid for the term of the contract, is foregone. No longer will the server be able to dictate the usage of its services by its ability to deny the acceptance of leases, nor will the clients be able to rely on the services that they request. Thus, the basic assumptions that parties participating in the system maintain while using it are no longer valid.

Conclusion

This paper has explored the fundamental differences between two state models, soft state and leasing. The use of soft state in the SNS system has been described to emphasize how the qualities of soft state are an ideal state mechanism for a cluster environment of non-reliable machines that need to maintain simple state and are all performing the same, highly parallelizable task. The BASE semantics of SNS are a perfect match for the relaxed consistency provided by soft state. Additionally, the design goal of simplifying maintenance while providing a high degree of fault tolerance is implicit in the soft state protocol.

In contrast, the benefits of a lease-based state mechanism were highlighted through a description of their use in the Jini architecture. In this case, where the goal is to provide highly heterogeneous services that join and leave the network frequently, a lease-based system provides the ideal state mechanism. Here, where there are possibly a large number of unique services, leasing reduces the amount of state maintenance messages that must be passed throughout the system, which is not scalable to a large number of hosts and services. Additionally, as each client requests a service, they can be assured that the service they are allocated will be available for the time they were allotted, with the quality guarantees they were promised.

These two systems focus on two different goals and choose the appropriate state schemes given the tasks they were trying to achieve. The discussion on the implications of switching the schemes shows that, by doing so, most of the goals that the systems were designed for have to be foregone. In the case of SNS with leases, the ease of managing a large cluster disappears. In the case of JINI with soft state, the scalability and assurances of service quality disappear.

Soft state and leasing are both important state management protocols that will have long futures in distributed computing research. However, their potential use in a system must be examined carefully to determine the trade-offs that each makes against the overarching goals of the proposed system. Judicious care should be taken in choosing one or the other, such that the overall system is not limited because of an incorrect state choice.

References

1. Bowers, K., Mills, K., and Rose, S. 2003. Self-adaptive leasing for Jini. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications*. 539-542
2. Clark, D. 1988. The design philosophy of the DARPA internet protocols. In *Proceedings on Communications Architectures and Protocols*. Stanford, CA.
3. Duvvuri, V., Shenoy, P., and Tewari, R. 2003. Adaptive leases: A strong consistency mechanism for the World Wide Web. *IEEE Trans. Knowl. Data Engin. 15*, 5. 1266-1276.
4. Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. 78-91.
5. Gray, C. and Cheriton, D. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*. 202-210.
6. Ji, P., Ge, Z., Kurose, J. and Towsley, D. 2003. A comparison of hard-state and soft-state signaling protocols. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Karlsruhe, Germany. 251-262
7. Ling, B. C., Kiciman, E. and Fox, A. 2004. Session state: Beyond soft state. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*. San Francisco, CA.
8. Lui, J. C. S., Misra, V., and Rubenstein, D. 2004. On the robustness of soft state protocols. In *Proceedings of the 12th IEEE International Conference on Network Protocols*. 50-60.
9. Ninan, A., Kulkarni, P., Shenoy, P., Ramamritham, K., and Tewari, R. 2002. Cooperative leases: Scalable consistency maintenance in content distribution networks. In *Proceedings of the 11th International Conference on World Wide Web*. Honolulu, Hawaii. 1-12.
10. Raman, S. and McCanne, S. 1999. A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 15-25.
11. Rose, S., Bowers, K., Quirolgico, S., and Mills, K. 2003. Improving failure responsiveness in Jini leasing. In *Proceedings of the DARPA Information Survivability Conference and Exposition*. 1-12.
12. Sharma, P., Estrin, D., Floyd, S., and Jacobson, V. 1997. Scalable timers for soft state protocols. *In Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies—Driving the Information Revolution*. 222-230.
13. Sun Microsystems. 1999. *Jini Technology Architectural Overview*.
14. Waldo, J. 1999. The Jini architecture for network-centric computing. *Comm. ACM 42*, 7. 76-82. **http://www.sun.com/software/jini/whitepapers/architecture.pdf**.

Biography

Dan Goldberg is a PhD student in computer science at the University of Southern California. His work in the USC GIS Research Laboratory focuses on spatial error of GIS processes and datasets, with particular regard to geocoding. He is a recipient of numerous DoD and Geospatial Intelligence scholarships and is the primary author of the North American Association of Central Cancer Registries' Geocoding Best Practices Guide.