Our interconnected world is increasingly reliant on distributed systems of unprecedented scale, serving applications which must share state across the globe. And, despite decades of research, we're still not sure how to program them! Modern global scale stretches existing abstractions—such as shared memory concurrency or transactional data stores—to the breaking point, pushing the task of successfully building against these abstractions ever deeper into the domain of experts. **My research focuses on building new systems and designing new language abstractions to make programming at scale feasible for more people, without sacrificing performance, correctness, or expressive power in the process**. In my work, I'm tackling distributed systems programming as both a programming languages *and* a systems problem: improving programmability of existing abstractions on one hand, and improving the scale at which we can offer these abstractions on the other. Truly addressing modern scale demands work *at the intersection* of these fields, requiring co-developed languages and systems that allow programmers to keep simple programming models while allowing distributed systems to operate at speed, without frequent coordination.

**Distributed Systems:** In core distributed systems, I show that new systems designs inspired by programming languages concepts can scale consistent replication to supercomputer speeds. With Derecho [4, 13], we push back against the notion that consistent replication can't keep up with modern data rates—while in turn taking inspiration from weakly consistent replication. My key insight was to rebuild Derecho's core message delivery protocol around monotonic observations, eliminating the need for blocking or locking on the critical path. Our results show that Derecho scales to the full speed of even infiniband networks, while exposing a convenient, zero-overhead API centered around replicated distributed actors.

**Language design:** Simply offering consistent replication is not enough. Existing consistent and concurrent abstractions—like shared memory—are made challenging by the potential for destructive data races. Yet shared memory is essential in real-world concurrent programs. In Fearless Concurrency [10], I present a type system that **provably eliminates** destructive data races in concurrent code without placing harsh limits on expressive power, ensuring programmers never need to debug a race on memory again. Crucially, my system is deployable in any C/Java-family language; in fact, it is currently under evaluation by Apple for inclusion into Swift.

While consistent replication is achievable at datacenter scale, it is more challenging to offer at global scale. Thus, many global-scale distributed systems continue to offer various forms of *weak consistency*, resulting in a zoo of confusing guarantees for users to navigate. Applications in this space must reason not just about a single consistency model, but often multiple overlapping consistency models simultaneously. Here, mechanisms that eliminate data races—like fearless concurrency or traditional transactions mechanisms—are not sufficient. In MixT [8], I take inspiration from the security literature to build an information-flow type system that ensures weak consistency is used responsibly. My approach successfully catches subtle bugs in example code, while remaining directly deployable atop existing transactional data stores.

**Research at the intersection:** MixT, Derecho, and Fearless Concurrency all stop at the same abstraction boundary: shared storage. While this abstraction offers an intuitive programming model, it makes a poor system interface; shared storage systems need to ensure consistency for *all possible uses* of the state they expose, often offering complex guarantees mismatched to what an individual application requires. Overcoming this limitation requires an integrated approach: we must design the language and system *together*. In so doing, we can build systems that adapt to the expressed needs of applications, paying the cost of strong consistency only when required for correctness. I demonstrate this principle by expanding on the key idea from Derecho—monotonic observations—in two projects: Hydro and Gallifrey. In Gallifrey [9, 7], I build a programming model in which programmers interact with shared, mutable, *multi-monotonic* objects. Despite offering strong consistency and race-freedom, Gallifrey can replicate monotonic objects under the weakest convergent consistency model—eventual consistency—with no risk to application correctness, requiring synchronization only for non-monotonic actions. In the wider Hydro project [2, 6], we combine Gallifrey's approach with insights from databases and security, finding new programming models that capture monotonicity without traditional shared memory or replicated objects. These programming models raise questions for future systems and languages research, which I am eager to explore.

### Scaling Consistent Replication with Derecho

Derecho is a library for building high-performance distributed code, supporting a *consistently-replicated actor* framework that Derecho calls "subgroups and shards." This abstraction is powerful and intuitive: users express programs as a group of services (or actors), each of which can be invoked via an external remote procedure call (RPC) and may in turn issue RPCs to other services within the system. The abstraction of a communicating replicated actor is difficult to offer with both low complexity and good performance. The main challenge is to ensure fault-tolerant, consistent replication at run-time. Modern industry-strength solutions tend to sidestep this challenge, often avoiding system-provided replication [14, 11] or weakening system-offered consistency [1]. With Derecho, we demonstrated that replication is possible at scale by rebuilding atomic multicast from the ground up, leveraging the emerging transport technology of remote direct memory access (RDMA) and rethinking traditional protocols in terms of RDMA-synchronized state. In particular, I rephrased virtual synchrony (a protocol for distributed consensus) in terms of loosely synchronized replicated tables, containing *monotonically growing* shared state. Boolean queries over monotonic state are stable: once they have returned true, they will *always* do so—even in the presence of racy, uncoordinated state updates. By phrasing protocols in terms of monotonic observations, I can safely eliminate blocking protocol steps and client-level locking—without sacrificing safety or liveness, and without weakening semantic guarantees. The resulting protocol achieves **strongly consistent replication among 16 replicas at 10GB/s**, nearly the line rate of our test cluster.

Replicated actors ensure that Derecho's insights reach the real world—in which not everything is monotonic. The API I built allows programmers to specify services as C++ classes, choose replication and resilience factors, and issue reliable RPCs between various services—all using the same syntax and tooling available for normal object oriented C++ code. While RPC among actors is common among industry-strength solutions[1, 14, 11], they often come with significant—and (according to their developers) unavoidable—run-time overhead. With Derecho, I show that through a combination of compile-time code and careful engineering, one can provide these abstractions with **no measurable overhead**.

### Bringing Fearless Concurrency to Usable Languages

Destructive data races in shared-memory concurrent programs have long been recognized as a challenge for programmers. **My type system for fearless concurrency [10] proves the absence of destructive races at compile time**, eliminating the need to monitor for, debug, or defend against such races at run time. The key idea presented in my PLDI paper [10] is to statically divide the heap into a tree of *regions*, which may be exchanged between threads. Objects may reference each other within the same region, or across region boundaries; the only requirement, enforced at function boundaries, is that cross-region links point to an *otherwise-unreachable* object subgraph. My type system deploys this invariant with zero run-time overhead and low user-exposed complexity: users need only annotate fields that may cross regions, leaving the rest to decidable type inference. For power users, I include the ability to describe complex region relationships, and feature a dynamic escape-hatch that is both easy to use and does not threaten correctness. Crucially, this system design is real-world ready: it can provide fearless concurrency in any C/Java family language. In reviewing my work for inclusion into Swift, the Swift language team specifically highlighted my system's ability to **represent common patterns with intuitive code**, its **low surface complexity**, its support for **separate compilation**, and its **extensibility** as **key improvements on Rust** and other competing industrial and academic languages.

In work currently under submission to PLDI, we take a radically different approach—exposing a *fully immutable* functional programming model to the user, while addressing the resulting overheads at compile-time. In this project, I solo-advised a team of four undergraduate students to craft a paper on lambda set specialization, a technique that eliminates the overhead of lambdas by statically determining all possible function targets of each call site and generating call-site-specialized versions of those functions. This project spans from **formal type system results** (complete with mechanization) to performance evaluations, demonstrating an **up to 6x improvement in higher-order functional ML code**.

### Mixing weak and strong consistency with MixT

At global scale, replicated datastores expose APIs with varying *weak consistency* guarantees, effectively exposing a trade-off between stronger consistency on the one hand and better performance on the other. However, this trade-off cannot be made once and for all; different data will require different consistency guarantees within one application—or even within one transaction. With MixT [8], we propose a model for *mixed-consistency* transactions, allowing programmers to safely manipulate data at various consistency models, all within the same transaction. We give a definition of safety for mixed-consistency transactions based on *noninterference*: changes to data stored at one consistency model should never affect data stored in a stronger model. This semantic condition captures not just direct influence—like copying a weakly consistent value into a strongly consistent store—but *all* forms of influence, including control flow. This condition *also* yields a transaction-splitting strategy; if data at weaker consistency cannot influence data at stronger, then we are *guaranteed* to be able to split a transaction by consistency model. Using this strategy, MixT is able to implement a *single* mixed-consistency transaction in terms of *multiple* single-consistency transactions, executed on any number of traditional data stores. Unlike similar type systems, MixT's transactions do not require explicit type annotations; instead, all annotations are inferred. Our results show that using mixed-consistency transactions can **significantly improve performance** vs. executing all transactions at only a single sufficiently-strong consistency model.

### Building at the intersection with Gallifrey and Hydro

The domain of large-scale geo-distributed systems programming is fertile ground for solutions that blend systems and programming languages insights. In the ongoing Gallifrey project [9], I'm looking for ways that programmers can build correct programs against simple, consistent abstractions—despite those programs being deployed in a globally-replicated, weakly-consistent ecosystem. In Derecho's core protocols, I leveraged a simple design principle: monotonic observations of ever-growing states. In Gallifrey, I built an entire programming model based around the idea of sharing *multi*-monotonic objects. The model is easy to explain: programmers write normal, sequential, object-oriented code—without worrying about race conditions or weak consistency. Threads (and processes) communicate by sharing sequential objects with *restricted interfaces*, which expose only a monotone set of operations. In this way, objects can be replicated under eventual consistency while still exposing a strongly-consistent, transactional interface to client programs—keeping both the replication system *and* the programming model simple and performant. In our VLDB paper [6], we argue that such a programming model is both usable and essential.

Were Gallifrey to focus only on simple monotonic shared objects, its programming model would match that of past work—like CRDTs (convergent replicated data types) [12], or LVars [5]. I go a step further, recognizing that a sequential object corresponds to not just one lattice, but one of several choices—each of which exposes a different subset of possible operations. By automatically *transitioning* between various convergent representations, Gallifrey objects are able to express even non-monotone operations, while still avoiding any synchronization outside of transition points. Taken together, Gallifrey's monotone restrictions and transitions ensure that the programming model of a Gallifrey application is expressive, performant, and safe.

At UC Berkeley, the Hydro Project—based on a vision I developed with professors Joe Hellerstein, Natacha Crooks, and Alvin Cheung [2]—combines my existing work with new insights from the databases and security communities. With Hydro, we're bringing the ideas behind Gallifrey, Derecho, and MixT to new paradigms, including program synthesis, query languages, streaming systems, and data storage systems. While this effort is still in its early days, it has yielded results: Katara [7].

I originated and advised the Katara project to fill a key need of Gallifrey: the ability to share sequential objects by restricting their interface. Doing this efficiently requires generating CRDTs that are *semantically equivalent* to shared sequential objects. Following the established pattern of *verified lifting* [3], Katara uses program synthesis to generate CRDT implementations of data structures from traditional single-threaded equivalents, finding encodings that **match the most efficient CRDTs in the literature**. Katara's key insight is to reframe verified lifting to require only a partial semantic match between the sequential object and the CRDT, allowing concurrent CRDT operations to be arbitrarily ordered.

## Future Work

In the Hydro Project, we're building out a vision to revolutionize the way cloud applications are built. Our 2021 CIDR paper [2] calls for the development of a new intermediate language for the cloud, capable of expressing existing application logic written in frameworks ranging from streaming systems to actor-based programs. This intermediate language needs to be relatively small and declarative, making it possible to leverage the techniques of Katara [7] to *automatically lift* existing framework-based code into equivalent declarative specifications. This technology would enable a wide variety of distributed programs to enjoy the established benefits of declarative code, from its capacity for whole-program optimizations to the ability to easily specify and verify correctness conditions. Fully realizing this vision will take many years; projects that we complete along the way have the potential for wide impact beyond the domain of distributed programming.

An opportunity for early impact lies in combining the insights of Gallifrey, Derecho, and Fearless Concurrency. In Gallifrey and Derecho, I take advantage of the fact that mutable objects may be safely shared concurrently so long as all updates and observations are monotonic. Integrating this insight into a system capable of *also* managing regions (as in Fearless Concurrency) would push such a system beyond the current holy grail of easy mutability recovery, and into the space of concurrent sharing with *mixed read-write access*—still without risking destructive data races. But this observation also raises questions: what *other* classes of mutability can we support without blocking? How should monotonicity be integrated into the language design—without exceeding our spare complexity budget? And most importantly, what real-world problems in concurrent programming would be solvable under such an analysis? Answering these questions would have wide-reaching impact across computer science: we have the chance to *nearly eliminate* the overhead of coordination and communication in applications ranging from modern distributed learning frameworks and scientific computing applications to auto-parallelizing compilers and mobile computing.

Such questions also arise in the space of weak-consistency and mixed-consistency programming. In MixT, we built a definition of mixed-consistency transactions, and an accompanying safety property based on noninterference. This definition is cogent and semantically sound, but it is too conservative; it fails to recognize that, under certain circumstances, one can *consistently observe* inconsistently-stored data. My work already demonstrates two ways to safely observe inconsistent data: via statically-tracked regions in Fearless Concurrency, and via monotonic updates in Gallifrey. Integrating these three efforts is a major undertaking; designing a type system which mixes both substructural and information-flow components with temporarily-restricted objects—while keeping things programmable and avoiding excessive complexity or annotation overheads—is fertile grounds for exciting research. These efforts also begin to map out what I believe to be a larger design space: there is daylight yet between the guarantees of full convergence provided by monotonicity, the guarantees of strong consistency assumed by most applications, and the guarantees of data-race freedom provided in fearless concurrency.

Regardless of how tightly we are able to integrate our languages and systems, general-purpose storage systems will still need a way to describe the consistency guarantees they make available. I believe that the wider question of how to correctly model consistency is not yet resolved; key to that is the question of what it *means* to observe potentially-inconsistent state. What are the *general mechanisms* by which inconsistent state can be consistently observed? How might we parameterize the set of consistent observations based on the guarantees offered by specific consistency models? In security, there are definitions for information flows that violate noninterference in controlled, acceptable ways. What might be the equivalent in consistency? These questions challenge a paradigm of shared-memory and distributed consistency that was established long before I was born. Answering them requires integrating insights from distributed systems, databases, formal consistency reasoning, type systems, and programming languages—and I believe that I am well-placed to work at—and across—these boundaries.

## References


[1] *Akka Documentation: Replicated Event Sourcing*. Nov. 2022. URL: https://doc.akka.io/docs/akka/current/typed/replicated-eventsourcing.html.

[2] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Mae Milano. "New Directions in Cloud Programming". In: *CIDR (Conference on Innovative Data Systems Research)* (2021).

[3] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. "Optimizing Database-Backed Applications with Query Synthesis". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 3–14. ISBN: 9781450320146. DOI: 10.1145/2491956.2462180. URL: https://doi.org/10.1145/2491956.2462180.

[4] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Mae Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. "Derecho: Fast State Machine Replication for Cloud Services". In: *ACM Trans. Comput. Syst.* 36.2 (Apr. 2019). ISSN: 0734-2071. DOI: 10.1145/3302258. URL: https://doi.org/10.1145/3302258.

[5] Lindsey Kuper and Ryan R Newton. "LVars: Lattice-Based Data Structures for Deterministic Parallelism". In: *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. 2013, pp. 71–84.

[6] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M. Hellerstein. "Keep CALM and CRDT On". In: *PVLDB* 16.4 (2023). DOI: 10.14778/3574245.3574268.

[7] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. "Katara: Synthesizing CRDTs with Verified Lifting". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563336. URL: https://doi.org/10.1145/3563336.

[8] Mae Milano and Andrew C Myers. "MixT: a language for mixing consistency in geodistributed transactions". In: *39th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. June 2018, pp. 226–241.

[9] Mae Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers. "A Tour of Gallifrey, a Language for Geodistributed Programming". In: *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 136. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 11:1–11:19. ISBN: 978-3-95977-113-9. DOI: 10.4230/LIPIcs.SNAPL.2019.11. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10554.

[10] Mae Milano, Joshua Turcotti, and Andrew C. Myers. "A Flexible Type System for Fearless Concurrency". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 458–473. ISBN: 9781450392655. DOI: 10.1145/3519939.3523443. URL: https://doi.org/10.1145/3519939.3523443.

[11] Philipp Moritz et al. "Ray: A distributed framework for emerging {AI} applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.

[12] Marc Shapiro. "A Comprehensive Study of Convergent and Commutative Replicated Data Types". en. In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. New York, NY: Springer New York, 2017, pp. 1–5. ISBN: 978-1-4899-7993-3. DOI: 10.1007/978-1-4899-7993-3_80813-1. URL: http://link.springer.com/10.1007/978-1-4899-7993-3_80813-1.

[13] Weijia Song, Mae Milano, Sagar Jha, Edward Tremel, Xinzhe Yang, and Ken Birman. "Derecho's Extensible, Intelligent Object Store". In: AISys@SOSP (2019).

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster computing with working sets". In: *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. 2010.