rNotify - A Scalable Application-Level Distributed Filesystem Notifications Solution

Ashwin Raghav Mohan Ganesh
Dept of Computer Science
University of Virginia
am2qa@virginia.edu

ABSTRACT

Filesystem (FS) event notifications are used my many applications to track file/directory objects. Although most operating systems (OS) are provisioned to notify applications of local FS changes [Table 1], there currently exist no scalable mechanisms to notify applications of non-local changes made to distributed FSs. In this paper, we propose rNotify, a scalable solution to this problem of notifying applications of events that occur on distributed filesystems. In particular, we focus on distributed notifications for scalable filesystems that provide global namespacing.

Our proposal has three claims. Firstly, the rNotify API will aim to provide application semantics nearly equivalent to the existing Linux notifications API. Linux applications can potentially use rNotify with minimal code changes. Secondly, by modifying the semantics of iNotify slightly, we claim that rNotify will dispatch event notifications with low latency for 98% of useful FS changes. Thirdly, our system will not be subject to event queue overflows and can support high Notification Throughput. We plan to support our claims by testing rNotify with GlusterFS that has linear horizontal scaling abilities with regard to performance and capacity.

1. INTRODUCTION

Prior to the introduction of FS event notifications, applications that required FS tracking polled the FS for changes. This mechanism was inefficient for the application as well as the FS being polled. Filesystem event notifications allow applications to track specific FS events that occur in directories of interest. In Linux, the iNotify kernel API is used by applications to subscribe to FS objects. An application interested in receiving notifications must subscribe to a directory or file, and specify the type of notification events that the application wants to receive.

Inotify [6] is an iNode based file notification Linux kernel API that is implemented at the Virtual File System (VFS) layer shown in Fig 1. Because all local FS changes made using POSIX API are made through the VFS layer, it is the ideal point to identify local FS mutations.

However, for a Distributed FS that implements only a stub on the client side, such a mechanism is insufficient. iNotify is unaware of events that are generated by other clients on the distributed FS's host machine.

rNotify is an out-of-band solution to the problem of monitoring files and directories whose locality is remote to the observing process. Since Distributed FSs are designed to scale in terms of capacity and performance, the scope of our problem extends beyond just locality. Our solution is hence scalable, location transparent, supports high notification throughput and ensures realiable failure notification.

Table 1: File System Notification API of Commercial Operating Systems

Operating Sytem	API
Linux Distributions	Inotify
OSX	fsevents
Windows	FindFirstChangeNotification
Solaris	File Events Notification(FEN)

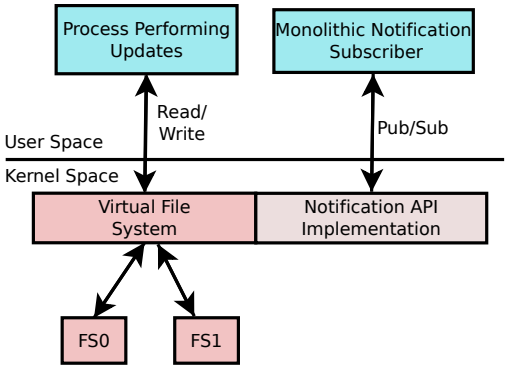

Figure 1: Monolithic Nature of Applications using Notification API.

2. MOTIVATION AND REQUIREMENTS

This section describes the various kinds of applications that currently benefit from FS notifications APIs. We also analyze the constraints and requirements that would impact potential solutions for providing distributed FS notifications. We explore other alternative solution designs and elaborate on why we chose to not use them.

2.1 A Case For File System Notifications

Many applications benefit from an efficient filesystem notifications API. Desktop search utilities are one such example. With the help of notifications APIs, desktop search utilities like Beagle [7] are able to reindex files that have recently been changed without scanning the entire filesystem

for deltas. Rather than polling all files in the relavent directories, these applications subscribe to directories via the FS notifications API, and are alerted by the kernel as soon as FS changes occur. This asynchronous mechanism allows applications to achieve change-to-reindexing times of less than a second.

Other applications of FS notifications include detecting changes in files and directories (e.g. configuration files, mail directories), guarding critical files, initializing automatic recovery, gathering file usage statistics, automatic upload handling, monitoring installations outside of packaging systems, automatic on-change backup and/or versioning and designing triggers for document databases.

By extending existing notifications mechanisms to accommodate distributed FS notifications, the functionality and performance of applications like desktop search programs can be significantly improved.

2.2 Requirements

Portability of Existing Applications: Existing applications that use iNotify currently either ignore distributed filesystems (like NFS or other equivalent protocols) or are simply not notified of remote changes[7] and must use polling or a nonscalable solution like the File Alteration Monitor (FAM) to monitor distributed filesystem events. It is required that a new solution be adoptable with minimal development effort.

Horizontal Scalability: Elasticity is the notion that an enterprise should be able to flexibly adapt to the growth (or reduction) of data and add or remove resources to a storage pool as needed, without disrupting the system. RNotify aims to provide such a notion.

Separation of Concerns: This requirement is closely tied to horizontal scalability. With a clear separation of concerns between components, the system design must ensure independent horizontal scalability of each component to improve specific requirements.

Performance: Since the solution continues to provide location transparency to applications, it is vital to provide low latency notifications. An increase in notification latency can result in degredation of the Application's performance. Although our primary focus is on delivering notifications to applications within a Local Area Networks (LAN), one of our goals is to support notification delivery via pragmatic multicasting that can be used effectively to deliver notifications across Wide Area Networks.

2.3 Design Alternatives

Before enumerating the design details of the rNotify system, we wish to mention three broad design alternatives that we considered for our system and their shortcomings. In the interest of brevity, we represent this graphically in Table 2 without further explanation.After exploring these design alternatives, we concluded that the notifications system would need to be out-of-band and at the application level.

Table 2: Design Alternatives and their Shortcomings

Design	Disadvantages
File System Modification	Incorporating varying FS data models, Modifying VFS API, Ensuring Backward Compatibility.
Modifying VFS	Breaks OS' s uniform way of treating File Systems
Modifying Inotify	Requires highly available listeners on client, imposed significant overhead on client, requires kernel patching.

3. RELATED WORK

The distributed systems community has put a lot of emphasis on the scalability of event notifications. We have drawn on this in our implementation and design. rNotify differs from existing work in three signficant ways.

Firstly existing client-side notifications broker implementations like FAM [4] are heavy on the client OS since brokers are run as high overhead OS daemons that need to be highly available to propagate notifications. Also, such a system was not built with the intention of catering to a truly scalable distributed FS.

Secondly, existing publish-subscribe systems are designed under the assumption that no single client will experience a high volume of notifications [1]. Our system targets a greater scale of notification density per client.

The third difference is our system's primary goal. A number of P2P notification and Publish systems like Bayeux [14], Scribe [12], and Siena [3] share our goal of providing a scalable notification delivery. These systems depend on an underlying overlay network to construct multicast trees and disseminate messages to Wide Area Networks. We, make no assumptions about underlying overlay networks. Since our focus is on improving performance within LAN, we adopt a low latency non hierarchical topology.

Our system is built without making any assumptions about the implementation of the underlying distributed filesystem protocols. This makes the rNotify system extendable to all filesystems that implement any transparent distributed filesystem protocols. Our provision for performance draws on practical experience with infrastructure as shown by other notification systems like Thialfi [1]. A substantial amount of prior work exists on publish - subscribe systems (e.g. [2, 10, 13, 11]), but these systems provide richer semantics and target lower scale systems.

4. DESIGN REQUIREMENTS

In this section we enumerate the different functionalities of any remote FS notification solution that can be retrofit to distributed FSs.

Multiplexing Subscriptions / Proxying: Linux systems impose a restriction on the total number of subscriptions allowed at the user level[1]. There is hence a need to multiplex subscriptions via a proxy at the File host. Also, an optimally performing file systems has ideally moved its' bottleneck to

[1]These numbers can be configured/changed by super users

the network. It is crucial to multiplex notifications that leave a File Host to conserve bandwidth.

Serialization: Once a notification for a file/directory mutation has been received by the system, it needs to be converted to a form that can be sent over the wire and conveniently deserialized at the client. This is a key requirement to support syntactic similarity to iNotify. Serialization has proved to be a process that is CPU bound.

Demultiplexing Notifications: Each relevant subscriber needs to receive the Serialized Notification. Without a multicast tree [2, 10, 13, 11], this requires copies of the notification to be sent to individual subscribers. As the number of subscribers increase, demultiplexing tends to be bound by the network bandwidth.

5. OVERVIEW

In this section we provide an overview of rNotify and it's overall architecture. We also analyse the semantic differences that applications need to accommodate in order to use rNotify effectively.

5.1 Comparison to Inotify

To support portability of applications using iNotify, our design achieves transparency in syntax to a large extent. Some differences in syntax and semantics are enumerated below.

Notification Burst Message In order to deliver useful notifications, clients need to understand a burst control mechanism that we incorporate. When a subscription experiences a burst of notifications (like when a file is being continuously appended to), there is little value in continuously informing an application of such mutations. We instead send a ENTRY_BURST notification to all relevant subscribers. An application that reseives a burst notification needs to conservatively assume that it will not receive notifications for the particular subscription for 'p' units of time. The assumption is conservative since the semantics we propose may resume notifications before 'p' units of time.

SystemUnavailable Message: The notion of reliability we provide requires that clients are made aware of system failures. When a client receives this message, it can invoke it's pollong path on the distributed FS.

Message Ordering: Unlike inotify, our semantics does not guarantee notification ordering. Other notification systems like Thialfi [1] have shown that ordering guarantees cannot be provided in a high performance notifications system. In our experience, we find that applications can be built without requiring a strong ordering in notifications delivery.

5.2 Architecture

As shown in Figure 2 the rNotify system is composed of five components: a client library, a Subscription Proxy, Dispatchers, Publishers and a Registry. Each of these are independent components that can be colocated or distributed and replicated. The separation of concerns between them is shown in Table 3

5.3 Client Library

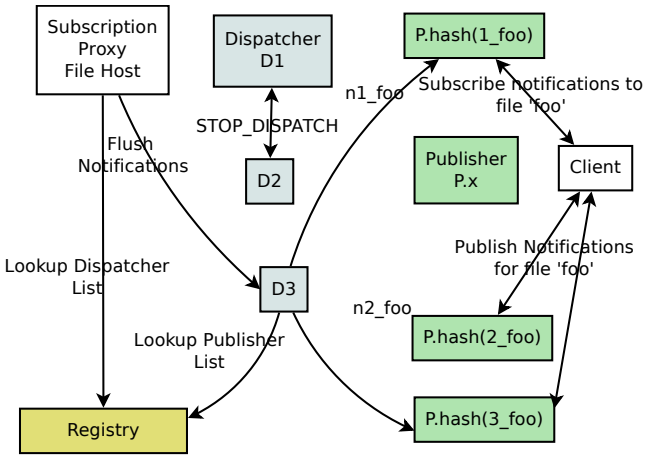

Figure 2: Overall Architecture

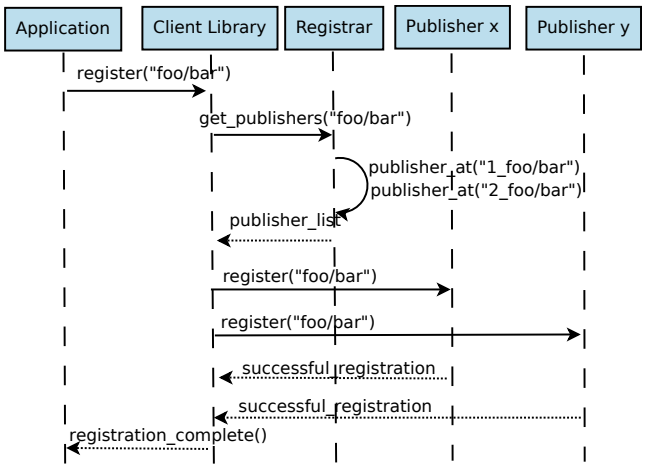

Figure 3: Subscription Sequence Diagram

```
interface subscriber{
    register(file_path);
    unregister(file_path);
}
interface listener{
    notify(file_descriptor);
    registration_failed(file_path);
    reissue_registrations(hashed_value_start, hash_value_end);
    reissue_registrations();
}
```

Figure 4: Client Library Interfaces

Table 3: Separation of Concerns

Component	Concerns
Client Library	Deserialize Notifications, Re-register after Publisher Replication
Subscription Proxy	Act as a proxy for subscribing clients
Dispatcher	Serialize Notifications, Control notification Bursts
Publisher	Demultiplex Notifications
Registrar	Track Publisher and Dispatcher replicas

Listing 4 shows the interfaces provided by the Client Library that applications are expected to implement. Clients can invoke a blocking call on the register method. Fig 3 indicates that when the application subscribes to a particular file path, the library contacts the *Registrar* and gets a list of all the Publishers where the subscription needs to be recorded. The client library registers the subscription with each of these Publishers. This two way communication helps ensure that a registrar's functionality is purely to return a list of Publishers using a method described in 5.6. The client has complete control of the subscription mechanism and is fully aware of when a registration is complete. Registering subscription to all Publishers is transactional from the application's point of view. Registration either succeeds or fails atomically.

The application also implements the listener interface. When a notification is received, the *notify* method is invoked and the application can consume notifications from the queue accessible via the parameterized file descriptor.

The polymorphic reissue registration operation serves two purposes. When a new Publisher is added, the client receives a reissue registration for files in a particular range on the *consistent hash* ring. In this case, the client computes the consistent hash on all its' subscriptions and renews the registration on files in that relevant range of the hash function. Secondly, when the application goes offline and returns either due to a network failure or due to planned behaviour, all subscriptions must be renewed using the non-parameterized variant of the *reissue registrations* operation.

5.4 Subscription Proxy

The subscription proxy (SP) resides on every File Host of the underlying Distributed File System. It's primary functionality is to multiplex subscriptions sent by the registrar.

The i/o thread shown in 5 receives subscriptions and invokes the iNotify *watch* API. In inotify, a watch is an idempotent API used to inform the Operating System about an object of interest. This thread is also responsible for reading blocks of data from the output buffer and flushing the notification blocks opportunistically to a radomly picked Dispatcher.

Avoiding Overflows: The proxy thread in 5 receives notifications on behalf of subscribers. Inotify semantics currently notifies subscribers by enquing messages onto a fixed size circular queue[2]. A subscribing entity can consume notifications by efficiently performing a *select* on the file descriptors that represent the queues. The semantics of the circular

[2]iNotify uses kqueue in FreeBSD and epoll in Linux

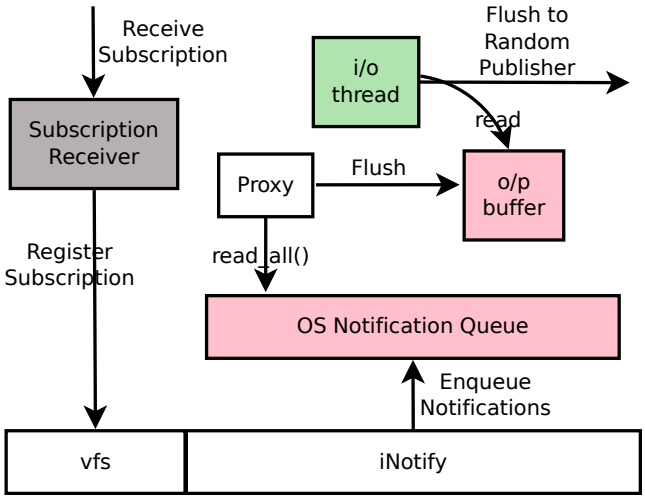

Figure 5: Subscription Proxy

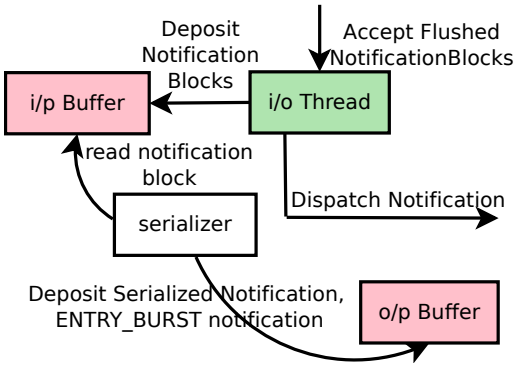

Figure 6: Dispatcher

queue require that applications consume notifications from these as fast as they can. Once the queue is filled, it is overwritten with Overflow messages. The key to avoiding queue overflows is to design a lightweight event-loop. We have verified that a simple *readall* operation on the queue and flushall operation to the output buffer can comfortably avoid overflows even at peak rates of activity on the ext4 filesystem. Any attempt to read notifications individually results in significant overflows.

5.5 Dispatchers

The Dispatcher is a replicable component that serializes the notification blocks sent by any *Subscription Proxy*. As mentioned in 5.4, every SP maintains state on the list of available Dispatchers provided to it on startup.

The i/o thread receives notification blocks sent by any SP and adds them to the input buffer. The i/o thread reads the output buffer and opportunistically forwards the packets to the addressed Publisher.

The serializer thread reads notification blocks from the input buffer, iterates through individual notifications converting each to a dispatchable, serialized form.

Detecting notification bursts: The *serializer* thread is also responsible for detecting notification bursts. By performing a

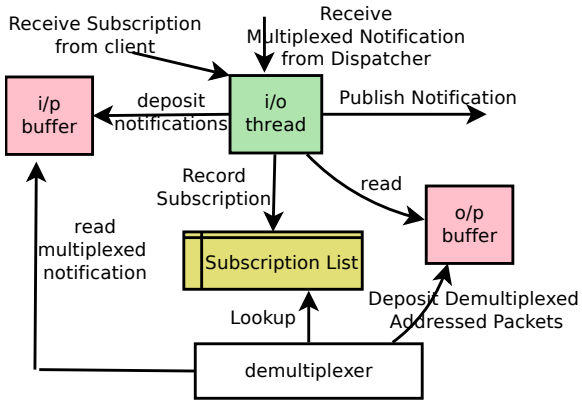

Figure 7: Notification Publisher

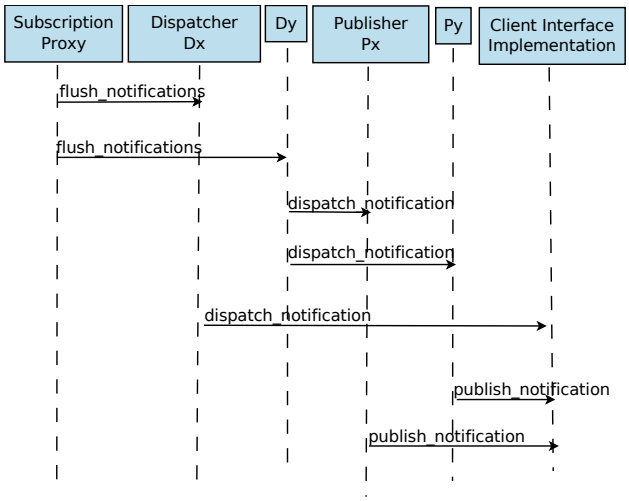

Figure 9: Notification Sequence Diagram

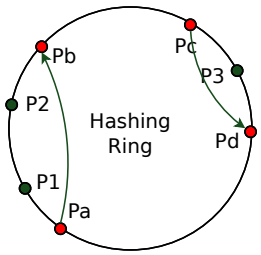

Figure 8: Relocation of Subscriptions

simple *frequency check* on the notification received, bursts are detected. On detecting a burst the io thread enqueues a ENTRY_BURST to the output buffer.

The semantics of the ENTRY_BURST message does not require that dispatchers be in synchrony. Each dispatcher makes an isolated decision based on its private frequency list. Those dispatchers that do not experience a burst of notifications for that particular subscription continue dispatching messages. The pessimistic semantics of the ENTRY_BURST notification help avoid the need to synchronize suspension of notifications across dispatchers.

Choosing a Publisher: The semantics of rNotify at the Client Library ensure that a client subscribes to a set of Publishers computed via an *consistent hash* function 'H' applied to the file path prefixed 'P' times with integers. An application may receive notifications from any of these P publishers. At the Dispatch stage, the same *consisten hash* function H is used to obtain the set of publishers with cardinality 'P'. The dispatcher randomly picks one of the publishers and forwards the notification. This acts as a simple load balancing mechanism without the need to communicate with the Publishers.

Replication: Dispatchers become CPU bound when a large number of objects under subscription are mutated. In our design, the Dispatcher is the most easily replicable component. All Dispatchers are symmetric in functionality and lock-free. When Dispatchers are added dynamically, SPs are notified via a separate control channel (not shown in 5). Once again the symmetric nature of these dispatchers ensures that no consistency is required in the Dispatcher list that each SP contains. An SP that has a stale Dispatchers List will not use the new Dispatcher.

5.6 Publishers

The publisher receives messages from Dispatchers, performs a lookup on the set of subscribers for a particular notification and demultiplexes the notification. Each Publisher demultiplexes notifications to clients for a set of files of cardinality 'F'. It is important to note that any client subscribing to any of these 'F' files may receive the notifications from other publishers returned to it by the *registrar* for the particular object. This design ensures that no single Publisher will become a bottleneck as a result of demultiplexing notifications for a highly subscribed file. We are also able to guarantee that notifications are delivered with at-most-once semantics.

Replication: A Publisher is bound by Network Bandwidth as the number of subscribers increases. In a non hierarchical scheme, it is crucial that this component remains replicable in order to scale the number of clients and subscriptions.

When a Publisher is added, all clients that subscribe to objects that are located at points on the *consistent hash* ring (shown in 8) that are to be handled by the new Publisher need to re-register their subscriptions. All online clients must complete re-registration before Dispatchers are made aware of the Publisher. Addition of a Publisher occurs is a two process.

Notifying Clients: As shown in 8, let us assume that the new Publisher occupies four virtual positions on the *consistent hash* ring. The set of clients that need to be notified to reissue registrations are those that subscribe to files whose positions on the ring are between - Pa and Pb, Pc and Pd. Our client side semantics require the invocation of the *register all* operation on these clients with the relevant ring positions as parameters. When the Publisher is added, the registrar enqueues a REGISTER message on the Publishers Pb and Pc. In order to avoid a sudden spike of subscription traffic from clients attempting to reissue registrations, it can enqueue these messages onto the Publishers at convenient intervals. This mitigates the risk of having a non-replicated registrar. After a Publisher demultiplexes a register message, it inform the registrar of the event. Once the registrar receives a confirmation from all relevant Publishers, the second phase can begin.

Notifying Dispatchers: Afer the registrar receives confirmations, the Dispatchers are asynchronously notified of the new Publisher. Notifying Dispatchers requires no synchrnization since a Dispatcher' s semantics expect only eventual consistency in Publisher List state. Once a Dispatcher is made aware of the new Publisher, no further configuration is required since it automati-

cally begins to appear on the *consistent hash* function space that the Dispatcher computes before a notification is dispatched. This elegant two step process is crucial to the system's ability to seamlessly replicate Publishers.

Such a two step process ensures that the entire operation is lock free. Also the registrar does not have the reponsibility of migrating subscriptions across publishers.

5.7 A note on Buffer sizes

It is noteworthy that all components of the system either have a buffer at the input and/or at the output. Consider a case where 1 million unique notifications that escape burst controllers are generated within a time interval 't'. If we were to assume that each notification were to occupy an average of 50bytes, the amount of data exchanged between the Subscription Proxy & Dispatchers as well as the Dispatchers & Publishers is 50 MegaBytes including the framing imposed by the rNotify protocol. By incorporating buffers as small as 50 MB at the input and output ends, a simple system with one File Host, three Dispatchers and two Publishers can contain 11 million undelivered notifications before beginning to overflow.

Tree Saturations: Once the queues in the system are filled, we expect the system to exhibit tree saturation that culminates in the SP being unable to flush its' buffers off to dispatchers. This results in a blocking call to *flush*. As a result, the File Host's Operating System Notification Queue overflows and iNotify overwrites *Overflow* notifications onto the queue. In this way, our system can guarantee Overflow notifications even when the system is subject to tree saturation.

5.8 Registrar

The registrar is the only non-replicable component of rNotify. A monolithic registrar greatly simplifies our design. This is acceptable since the registrar has two functions both of which are infrequent events in the life-cycle of our system.

The registrar is responsible for receiving subscriptions from clients, computing the *elastic hash* function and returning the list of Publishers for the subscribed object. We believe that the volume of subscription traffic will be condiderably smaller than the notification traffic over the system's lifespan. It is fair to assume that all applications will not arrive simultaneously.

It is also the starting point for the propagation of replication configurations. When Dispatchers are replicated, the registrar asynchronously notifies all Subscription Proxies. As mentioned in 5.6, it carries out a two step update to notify Clients and Dispatchers when Publishers are replicated.

5.9 Design Aspects not covered in this section

In the interest of brevity, we have not covered some other sub component design schemes. We hope to cover these in detail in our post-completion paper. Aspects that have not been covered include Abstract Data Types provided to the Client Library, Decoupled Garbage Collection, High Availability Dynamo like [5] data store to persist subscription lists.

6. FAILURE MODES

In the interest of brevity, we have represented the failure modes that our reliability guarantees support in Table 4.

Table 4: Supporting Component Failures

Failing Component	System Reaction
Subscription Proxy Failure	When the File Host is restarted, SP recovers simultaneously
Dispatcher	SP flushes messages to alternate dispatchers when no Ack is received
Publisher	Notifications are dispatched to other Publishers. Clients are unaware.
Slow Clients	Client Library throws a custom expection
Dropped TCP Packets	Not supported

7. EVALUTATION

Horizontal Scalability - The Purpose: File System notification API as implemented in the POSIX standard is essentially a producer-consumer scenario. The designers of the API have drawn on practical experience that shows that File System Activity shows short periods of spiked activity. There have been other moderately successful attempts at Mathematically modelling File System Activity[8]. We plan to demonstrate that Dispatchers can be replicated to accomodate increased File System throughput. We also plan to show that Publishers can be replicated to increase notification throughput. Our results will hence help system designers use simple queueing theory models as in Fig 10 to design replication strategies for known levels of File System Throughput.

8. EXPERIMENTAL SETUP

Choice of Filesystem: We plan to evaluate rNotify using a deployment of the Gluster File System. GlusterFS is an open source, distributed file system capable of scaling to several petabytes and handling thousands of clients. GlusterFS clusters together storage building blocks over Infiniband RDMA or TCP/IP interconnect, aggregating disk and memory resources and managing data in a single global namespace. GlusterFS is based on a stackable user space design and can deliver exceptional performance for diverse workloads.

Apart from it's performance characteristics, an important reason we chose Gluster for evaluation is because of it's server implementation. Unlike filesystems like Lustre [9], Gluster does not bypass the VFS on the File Host and can optionally avoid file striping. Our subscription proxy design should hence work seamlessly fit into the File Host without having to patch the Gluster FS.

Deployment Strategy: We plan to perform our tests on Amazon EC2's Cluster family of instance types optimized for HPC applications. Cluster instances can be launched within a Placement Group. All instances launched within a Placement Group have low latency, full bisection 10 Gbps bandwidth between instances. Like other Amazon EC2 resources, Placement Groups are dynamic and can be resized. Cluster Compute and Cluster GPU instances specify the underlying processor architecture in their definition to enable developers to tune their applications by compiling for that specific processor architecture in order to achieve optimal performance.

At this point, we are unable to comment on the values ranges of the control variables since these need to be determined experimentally in a virtalized environment.

File System Workload We have considered using the PostMark and the Parallel Postmark benchmark. We are yet to devise a strategy on how to simulate bursts of File System activity. We plan to obtain real FS traces and have started work on obtaining these from the popular music streaming site Pandora that uses GlusterFS.

8.1 Measurement

Latency In order to demonstrate rNotify's ability to deliver notifications at a nearly constant latency, we plan to vary GlusterFS throughput and measure the median latency from when a File System event occured to when the relevant notification leaves the Publisher. Increasing the number of subscription will exercise our Publisher. By fixing the number of clients and number of

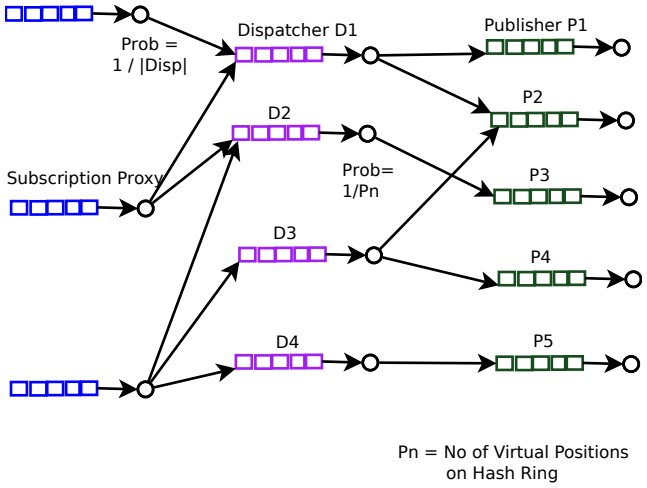

Figure 10: Simple Queueing Model

subscriptions per client, we plan to perform the experiment at various degrees of Dispatcher replication in order to demonstrate nearly Constant Latency and the effectiveness of burst control.

Notification Throughput We wish to demonstrate rNotify's ability to scale Notification Throughput (NT) in two scenarios. By fixing the File System Throughput and varying the number of subscriptions, we plan to measure NT at varying degrees of Publisher replication. In a similar setup, we plan to measure NT by varing the number of subscribers.

Self Imposed Delays We expect that a substantial portion of the latency measured is a result of the self imposed batching delays of rNotify. We plan to measure the average contribution to notification delay by each component of the system. Depending on the performance observed, we hope to fine tune notification delays and potentially show better performance at higher resource consumption rates.

Resource Utilization We plan to measure the resource utilization of the individual components at varying degrees of subscription and clients. We are yet to devise a strategy on how the resource consumption of distributed components in a virtualized environment can be measured. We hope to address this in our paper.

9. FUTURE WORK

Complete Message Ordering: We wish to improve the application semantics of our system and bring it closer to inotify by guaranteeing ordering. We foresee that this will be be hard to implement without impacting the performance of the entire system.

Security: In our current scheme, we have no security model and the scheme is susceptible to eaves dropping. We hope to design a security model that can work seamlessly with authorization mechanisms that distributed filesystems and operating systems support.

10. CONCLUSION

We have presented rNotify as a service that can be used by applications using iNotify to receive file notifications from distributed filesystems. Internally we plan to use a combination of soft-state on the server, asynchronous dispatch, asynchronous replication and reliable notification of failures to help distributed filesystem providers seamlessly integrate rNotify into their systems. We plan to achieve these things without requiring severe application rewrites and infrastructure revamps. We also plan to test rNotify in a High Performance Cluster to confirm results that indicate good scalability and performance. We hope that future Linux OSs will be shipped with rNotify as an out-of-band solution for remote FS notifications.

11. ACKNOWLEDGEMENTS

Several people provided valuable insights during the design stage. In particular we wish to thank Tommy Tracy and Sharadha Ramakrishnan for their contributions.

12. REFERENCES

[1] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: a client notification service for internet-scale applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 129–142, New York, NY, USA, 2011. ACM.

[2] Y. Amir and J. Stanton. The spread wide area group communication system.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, Aug. 2001.

[4] S. G. I. Corp. Fam overview.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[6] R. L. John McCutchan and A. Griffis. Inotify requirements and documentation, 2005.

[7] N. F. Jon Trowbridge, Robert Love and D. Camp. The beagle desktop search project, year = 2009, url = https://help.ubuntu.com/community/Beagle,.

[8] N. Joukov, T. Wong, and E. Zadok. Accurate and efficient replaying of file system traces. In *In Proc. USENIX Conference on File and Storage Technologies (FASTâĂŹ05*, pages 337–350. USENIX Association, 2005.

[9] P. Koutoupis. The lustre distributed filesystem. *Linux J.*, 2011(210), Oct. 2011.

[10] P. R. Pietzuch and J. M. Bacon. Hermes: A distributed event-based middleware architecture. pages 611–618, 2002.

[11] R. V. Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Comm. of the ACM*, pages 76–83, 1996.

[12] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, NGC '01, pages 30–43, London, UK, UK, 2001. Springer-Verlag.

[13] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *In Proceedings of the International Symposium on Software Reliability Engineering*, 1998.

[14] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '01, pages 11–20, New York, NY, USA, 2001. ACM.