Análisis del comportamiento de aplicaciones paralelas y distribuidas por medio de técnicas de emulación de redes

Geier, Maximiliano Iván

2018

UNIVERSIDAD DE BUENOS AIRES
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Análisis del comportamiento de aplicaciones paralelas y distribuidas por medio de técnicas de emulación de redes

Tesis presentada para optar al título de Doctor de la
Universidad de Buenos Aires en el área Ciencias de la Computación

Lic. Maximiliano Iván Geier

Director de tesis: Dr. Esteban Mocskos
Consejero de estudios: Dr. Diego Garbervetsky
Lugar de trabajo: Departamento de Computación, Facultad de Ciencias Exactas y Naturales

Buenos Aires, noviembre de 2018

Fecha de defensa: 13 de diciembre de 2018

Firma

Análisis del comportamiento de aplicaciones paralelas y distribuidas por medio de técnicas de emulación de redes

Resumen

Una gran cantidad de aplicaciones paralelas se encuentran programadas utilizando *Message Passing Interface* (MPI), que funciona como un standard *de facto* en el mundo de la computación de alto rendimiento. Por otro lado, los paradigmas de *Fog* y *Edge Computing* emergieron como una solución a las limitaciones del modelo de Cloud Computing para servir a una gran cantidad de dispositivos eficientemente. Estos últimos cuentan con un poder de cómputo inutilizado que puede ser explotado para ejecutar aplicaciones paralelas. Nos focalizamos en la siguiente pregunta: ¿Pueden las aplicaciones basadas en MPI aprovechar el incremento en los recursos disponibles distribuidamente por medio del paradigma de Fog/Edge Computing?

En este trabajo presentamos SherlockFog, una herramienta para experimentar con aplicaciones paralelas en configuraciones de red arbitrarias. Proponemos una metodología para estudiar si es factible ejecutar aplicaciones paralelas en entornos Fog/Edge. Estudiamos la indicidencia del empeoramiento de las condiciones de red en diversos *benchmarks* de la versión paralela MPI de los *NAS Parallel Benchmarks* en topologías de red fog.

Adicionalmente, proponemos una extensión a SherlockFog que hace uso de la herramienta Intel Pin para inyectar instrucciones de manera paramétrica en el código a ser estudiado, imitando procesadores con diferente poder de cómputo. Analizamos el impacto de nodos más lentos en dos *benchmarks* y mostramos que la incidencia de un único nodo más lento es significativa, pero incorporar nodos adicionales más lentos no acentúa dicha degradación. El efecto de la latencia también es analizado, pero su impacto depende del patrón de comunicación del código evaluado.

Finalmente, mostramos que nuestra metodología también es aplicable al estudio de otros tipos de sistemas distribuidos. Utilizando uno de los clientes oficiales de la criptomoneda Ethereum, reemplazamos el algoritmo de minado con un modelo simulado construido en base a las características estadísticas del proceso real, e instrumentamos el cliente para capturar eventos de red de interés. Propusimos escenarios de red de diversos tamaños en los cuales estudiamos la incidencia del tiempo de *target* en la presencia de *forks* en la red. Mostramos que, incluso utilizando una plataforma experimental de hardware convencional, es posible utilizar nuestra herramienta para estudiar la dinámica de sistemas basados en *blockchain* de hasta cientos de nodos.

Analysis of the Behavior of Parallel and Distributed Applications using Network Emulation Techniques

Abstract

A large number of parallel applications are programmed using Message Passing Interface, which is a de facto standard in High Performance Computing environments. On the other hand, the Fog and Edge Computing paradigms have emerged as a solution to the limitations of the Cloud Computing model to serve a huge amount of connected devices efficiently. These devices have unused computing power that could be exploited to execute parallel applications. We focus on the following question: Can MPI-based applications take advantage of the increasing number of distributed resources available through Fog/Edge Computing Paradigm?

In this work, we present SherlockFog, a tool to experiment with parallel applications in arbitrary network setups. We propose a methodology to study the feasibility of running parallel applications in Fog or Edge environments. We study the effect of worsening network conditions for several benchmarks of the MPI version of NAS Parallel Benchmarks on fog-like network topologies.

Further, we propose an extension to SherlockFog that makes use of the Intel Pin Tool to inject instructions parametrically in the target code, mimicking CPUs with different computing power. We analyze the impact of slower nodes on two benchmarks and show that the incidence of a single slower node is significant, but slowing more nodes down does not further degrade performance. The latency effect is also analyzed, but its impact depends on the communication pattern of the target code.

Finally, we show that this methodology is also useful to study other types of distributed systems. Using one of the mainstream clients of the Ethereum cryptocurrency, we replaced the mining algorithm with a simulation model built upon the statistical characteristics of the mining process and instrumented the client to capture relevant network events. We propose several network scenarios of increasing size in which we study the incidence of the target time in the presence of forks in the network. We show that even using a small testbed consisting of just commodity hardware, it is possible to use our platform to study the dynamics of blockchain-based systems up to hundreds of nodes.

To those who have encouraged
me to come this far

Contents

Part I

Resúmenes en castellano

1 Introducción

El proceso de desarrollo se apoya sobre el uso de diferentes herramientas, tales como editores especializados, herramientas de *debugging*, *profilers* y *frameworks* para *testing* [1]. Dichas herramientas suelen ser ejecutadas en la computadora del desarrollador. Los sistemas distribuidos no son diferentes, pero suponen desafíos adicionales ya que el entorno de ejecución en producción se extiende sobre diferentes sistemas. Las aplicaciones paralelas son un tipo particular de sistema distribuido que comprende múltiples procesos ejecutados en diferentes unidades de cómputo sobre uno o más sistemas. El desarrollo de entornos especializados, tales como los *clusters*, fue guiado por la necesidad de ejecutar este tipo de aplicaciones más eficientemente.

El tamaño y la complejidad de los sistemas distribuidos tuvieron un importante incremento en las últimas décadas debido a la proliferación de tecnologías instaladas sobre Internet y otras redes. Algunos ejemplos incluyen servicios de gran escala sobre Internet, cómputo ubicuo, Cloud, Internet of Things (IoT), sistemas de almacenamiento, redes de sensores y nodos móviles [2]. Esta complejidad es reconocida como un problema muy importante, ya que muchos sistemas que se encuentran actualmente en funcionamiento se vuelven difíciles de administrar, mantener y cambiar, resultando en costos de desarrollo elevados para las organizaciones. Es por este motivo que se vuelve necesario construir herramientas para entender este tipo de sistemas.

Esta tesis se focaliza en el estudio de sistemas distribuidos y paralelos en entornos heterogéneos, teniendo en cuenta tanto la conectividad de la red como la heterogeneidad de CPUs. Para este fin, desarrollamos una herramienta llamada SherlockFog, descripta en el capítulo 2, y proponemos una metodología para el estudio de sistemas distribuidos, focalizándonos particularmente en aplicaciones paralelas que utilizan una implementación de MPI para resolver problemas de cómputo científico. Luego, presentamos en el capítulo 3 diversos escenarios inspirados en el paradigma de Fog/Edge Computing para estudiar cómo se modifica la *performance* en varias aplicaciones características. En el capítulo 4, extendemos la propuesta anterior para estudiar aplicaciones paralelas en entornos IoT, dotando a SherlockFog de un mecanismo para modelar CPUs más lentas.

En el capítulo 5 proponemos que esta metodología puede ser utilizada para estudiar sistemas distribuidos tales como criptomonedas basadas en la tecnología de *blockchain*, mostrando que SherlockFog puede ser utilizado para analizar el protocolo de la criptomoneda Ethereum en un escenario controlado.

Para finalizar, concluimos y mostramos posibles líneas de trabajo a futuro en el capítulo 6.

1.1 Estudio de los sistemas distribuidos

Los sistemas distribuidos se presentan en muchas áreas de la computación, desde las aplicaciones científicas hasta sistemas de distribución de contenidos. Muchos de estos sistemas son altamente heterogéneos, donde diferentes subsistemas y tecnologías interactúan al mismo tiempo usando protocolos estandarizados. Existen iniciativas desde hace varias décadas para evaluar las propiedades de estos sistemas, tales como confiabilidad, capacidad de recuperación, *performance* o seguridad. En muchos casos, los investigadores recurren a la experimentación: analizan el comportamiento de un sistema ejecutándolo en un escenario determinado y capturando los datos que puedan ser de interés. Podemos categorizar el trabajo experimental en sistemas distribuidos en los siguientes tres paradigmas [3, 4]

- **Simulación**: un prototipo o modelo de la aplicación es ejecutado sobre un entorno modelado. La confiabilidad de los resultados depende de la validez de los modelos utilizados.
- **Emulación**: la aplicación real es ejecutada sobre un entorno simulado. El entorno puede ser controlado utilizando técnicas de virtualización, o por medio de la inyección de una carga artificial en recursos reales como la red o las CPUs. La principal ventaja de este enfoque es la posibilidad de analizar cambios en una aplicación o el paquete de software utilizado.
- **Experimentación**: la aplicación real es ejecutada sobre el entorno real. Aunque deseable, no siempre es posible utilizar este enfoque, ya que requiere acceso a una plataforma instrumentada que se corresponda con el entorno real. Esto puede ser prohibitivamente caro o no estar disponible. Un problema adicional es mantener un entorno controlado sobre Internet, ya que el tráfico adicional puede inducir a conclusiones incorrectas y resultados irreproducibles.

Este trabajo se focaliza en el paradigma de la **Emulación**, ya que creemos que es el enfoque más apropiado para estudiar sistemas que se encuentran en desarrollo o que continúan evolucionando.

1.2 El modelo de Fog/Edge Computing

En los últimos años, el modelo de Cloud Computing emergió como una alternativa a adquirir y manejar una infraestructura de cómputo propia. Cloud Computing permite utilizar recursos según las necesidades del usuario, dándole a este último la flexibilidad de pagar solamente por lo que utiliza. Esta tecnología soluciona de manera eficiente distinto tipo de aplicaciones, tales como los servidores *web*, bases de datos, almacenamiento y procesamiento en bloques. Aprovechando la economía de escala de los grandes centros de cómputo, también resulta más barato para el usuario final.

La técnica utilizada para delegar parte del cómputo a un sitio remoto se conoce como *offloading*. La misma permite incrementar el uso de recursos como se requiera, en base a la complejidad de la tarea.

Debido a que la Cloud se encuentra generalmente lejos de los clientes, aplicaciones que son sensibles a la latencia pueden sufrir una degradación en su *performance* en esta configuración. Bonomi *et al.* [5] definen el modelo de Fog Computing como una plataforma altamente virtualizada que provee servicios de cómputo, almacenamiento y red entre dispositivos que consumen recursos de la Cloud y la propia infraestructura. Este modelo permite que algunos servicios puedan aprovechar la técnica de *offloading* sobre los clientes de la Cloud, y de esta manera reducir los requerimientos de latencia, y al mismo tiempo mejorar la elasticidad del sistema. Este último término se define como la capacidad de adaptarse a cambios en la cantidad de trabajo [6]. Yendo más allá, la proliferación de dispositivos IoT con un poder de cómputo cada vez mayor le ha dado entidad a los nodos en el borde (*edge*) de la red, resultando en el modelo de Edge Computing [7]. Estas configuraciones deben soportar movilidad, geodistribución, una baja latencia y tiempo limitado de utilización [8]. En este modelo, los nodos en el borde de la red cooperan entre ellos para obtener resultados en un tiempo de respuesta menor o con menor uso de ancho de banda, al mismo tiempo que se mejora la seguridad de los datos y la privacidad con respecto a resolver este procesamiento en la Cloud.

1.3 MPI

La API estándar que se utiliza en aplicaciones paralelas de cómputo científico es MPI [9, 10, 11]. La misma fue diseñada para proveer una abstracción que maneje el intercambio de datos y la sincronización entre procesos, sin importar si están localizados en el mismo nodo o en máquinas diferentes.

MPI es una API muy poderosa para desarrollar programas paralelos, que además funciona a muy bajo nivel. Paralelizar una aplicación—transformar un programa *serial* en uno paralelo utilizando MPI—involucra varios pasos como cambiar y/o replicar estructuras de datos, distribuir datos explícitamente entre procesos, recolectar resultados y sincronizar procesos. Por este motivo, MPI es considerado el "lenguaje ensamblador" de la programación paralela. Los pasos mencionados no son incrementales, requiriendo en ciertos casos la reescritura total del programa original.

Existen varias implementaciones que cubren parcial o totalmente el estándar de MPI en alguna versión, y que ofrecen distinto soporte de protocolos de transporte de red. En su mayoría, se focalizan en clusters High Performance Computing (HPC), suponiendo que los nodos tienen conectividad total entre ellos. Además, son sensibles a cambios en la red, pudiendo detenerse el cómputo por completo si tan solo un nodo se desconecta o cambia su ubicación en la red[1].

La primera implementación abierta del estándar MPI-1 fue MPICH, del Argonne National Laboratory. Adicionalmente, LAM/MPI y, luego, Open MPI ofrecen alternativas de código abierto. MPICH sirve como base para varias implementaciones de MPI, y sigue siendo mantenida en el año 2018, soportando la versión 3.2 del estándar.

También existen implementaciones comerciales de varias compañías, incluyendo HP, IBM, Intel, Cray, Microsoft y otras que han participado en el mundo HPC.

Las operaciones básicas que define MPI pueden ser categorizadas según orígenes y destinatarios de la comunicación como uno-a-uno, uno-a-muchos, muchos-a-uno y muchos-a-muchos. Uno-a-uno o **Point-to-point (P2P)** incluye enviar y recibir mensajes desde y hacia un único proceso. Esto puede ser resuelto sincrónica o asincrónicamente dependiendo de las necesidades del usuario. Las operaciones uno-a-muchos, muchos-a-uno y muchos-a-muchos (o **collective**) son utilizadas para propagar datos a otros procesos y recolectar resultados parciales. Otras operaciones de este tipo que son de uso común son las barreras de sincronización o el envío múltiple de mensajes uno-a-muchos o muchos-a-uno en paralelo en todos los procesos.

1.4 NAS Parallel Benchmarks

En 1991, el programa Numerical Aerospace Simulation (NAS) del NASA Ames Research Center define un conjunto de *benchmarks* como parte de un programa interno para evaluar supercomputadoras paralelas [13]. Esto era de suma importancia para que NASA pudiera decidir adecuadamente qué nueva supercomputadora comprar, ya que la información técnica provista por vendedores y la comunidad científica muchas veces sobreestimaba las capacidades de ese tipo de hardware. Inicialmente y al no existir una forma estandarizada de programar aplicaciones paralelas cuando se comenzó con este trabajo, los *benchmarks* fueron diseñados en "lápiz y papel", es decir, eran un conjunto de requerimientos en un documento técnico que describían, precisamente y de una manera estándar, los algoritmos y los datos de entrada necesarios. A medida que avanzaban las distintas tecnologías, fueron apareciendo implementaciones específicas para MPI, OpenMP

[1]Con la excepción de FT-MPI [12].

y otros modelos de programación paralela. En este trabajo llamaremos indistintamente NPB o NPB-MPI a la implementación de los *benchmarks* sobre MPI.

Los NAS Parallel Benchmarks (NPB), en su formulación original, consistían de 8 problemas, cada uno focalizado en un problema específico del ámbito del cómputo científico. Los mismos proveen una buena base para estudiar distintos aspectos de las aplicaciones HPC, dado que ya han sido estudiados extensamente por la comunidad [14, 15, 16, 17, 18] en distintos contextos. Cinco de ellos son "kernels" (EP, MG, CG, FT e IS) y los últimos 3 son aplicaciones de Computer Fluid Dynamics (CFD) (LU, SP y BT). Los kernels son problemas compactos que se focalizan en un tipo particular de cómputo numérico. Las aplicaciones de CFD reproducen el movimiento de datos y el cómputo necesario para resolver problemas en dicho dominio y en otras simulaciones físicas en 3D.

Cada *benchmark* puede ser instanciado en un tamaño de problema particular (clase) y cantidad de nodos (tamaño). Cada clase predefine el tamaño del problema según su valor y de qué *benchmark* se trate.

2

Metodología

La metodología propuesta en esta tesis está fuertemente asociada a una herramienta novedosa llamada SherlockFog, que presentamos en las siguientes secciones. Exploraremos las características, casos de uso y limitaciones de la misma a lo largo de este capítulo.

2.1 Propuesta

Proponemos una nueva metodología para el análisis y conversión de aplicaciones distribuidas al paradigma de Edge Computing. Nuestra propuesta se focaliza en el estudio del impacto de distintos patrones de comunicación en las aplicaciones, haciendo énfasis particularmente en MPI, ya que es la API más utilizada para implementar cómputo científico distribuido utilizando pasaje de mensajes.

Desde una perspectiva científica, dada una aplicación y uno o más escenarios de uso (red y configuración), el usuario puede definir la configuración programáticamente y evaluar cada una utilizando nuestra herramienta. Podemos diferenciar los siguientes pasos:

1. El usuario elige una aplicación y una topología y construye un *script* que define un el experimento en SherlockFog. Dicho *script* utilizará un conjunto de nodos físicos especificados por el usuario.

2. SherlockFog es ejecutado en el coordinador, el cual se conectará a cada nodo para inicializar la red virtual.

3. Se generan enlaces virtuales correspondientes a la topología descripta en el *script* de entrada. Se utiliza *ruteo* estático para permitir la comunicación entre cada interfaz de red virtual.

4. El código de cada aplicación es ejecutado en los nodos virtuales según lo que haya especificado el usuario en el *script* de entrada.

SherlockFog permite modificar los parámetros en tiempo de ejecución, y repetir los experimentos con distintos parámetros o topologías. La salida puede ser recolectada para su posterior análisis.

2.1.1 Características principales

Desde la perspectiva del usuario, las características principales son las siguientes:

- SherlockFog puede ser ejecutado en hardware convencional.

- Los *scripts* son escritos en un lenguaje propio denominado *fog*, que permite definir la topología de red, parámetros de los experimentos y comandos para ejecutar la o las aplicaciones.
- La herremienta puede ser utilizada sobre un único nodo físico o sobre varios, permitiendo escalar la red emulada sobre otros recursos.
- La red, CPU y la memoria pueden ser compartimentadas entre nodos virtuales que se instancian sobre un mismo sistema utilizando técnicas de virtualización liviana.
- El código de usuario se puede ejecutar sin ningún tipo de modificación, permitiendo la evaluación de programas tanto de código abierto como cerrado.
- La movilidad puede ser modelada cambiando el ancho de banda o la pérdida de paquetes de un enlace en tiempo de ejecución. Esta característica es útil para modelar entornos Edge o Fog.

2.1.2 Consideraciones sobre el uso de la herramienta

Presentamos algunas consideraciones a tener en cuenta en el uso de la herramienta:

- No es posible experimentar con tráfico *multicast* debido al mecanismo utilizado para *rutear* el tráfico al nodo virtual correspondiente. Sin embargo, nuestro foco es las aplicaciones MPI, que utilizan múltiples mensajes *unicast* para resolver las comunicaciones globales.
- El ancho de banda total de todos los enlaces virtuales es compartido con el o los medios físicos utilizados. Es posible tener un mayor control de este problema limitando el ancho de banda asignado a cada enlace virtual.
- La latencia real de cada enlace físico debe ser tenida en cuenta al diseñar un experimento. Como ocurre en todas las herramientas que usan redes emuladas, si la latencia virtual deseada es de un valor muy próximo a la latencia real del enlace subyacente, puede haber imprecisiones en los resultados obtenidos.

2.1.3 Emulación de la *performance* de plataformas IoT

Las técnicas tradicionales para reducir el ancho de banda de CPU disponible para un proceso permiten dotar al sistema de una cota superior de uso para reducir variaciones en la latencia percibida. Sin embargo, dichos mecanismos no son los adecuados para emular la *performance* de dispositivos mucho más lentos, tales como los encontrados en plataformas IoT. Este tipo de plataformas presenta arquitecturas, *performance* de CPU y características muy diversas a las encontradas en sistemas HPC o de hardware convencional. Para poder reproducir el cómputo fielmente, SherlockFog debe poder emular la *performance* de los mismos, ya que ésta última afecta el solapamiento entre cómputo y comunicación. Esto nos lleva a proponer una solución por medio del uso de una herramienta externa para enlentecer el cómputo. La herramienta que utilizamos es Intel Pin Tool [19].

Definimos un módulo para Pin, llamado `slowdown`, que inserta un número fijo, configurable, de no-instrucciones[1] antes de la ejecución de cada instrucción del programa a evaluar.

Pin es un compilador Just-in-time (JIT), por lo que las instrucciones adicionales son agregadas en tiempo de ejecución. El número de instrucciones es ajustable por proceso a través de archivos de configuración, permitiendo modelar plataformas con poder de cómputo heterogéneo.

Un incremento en el número de no-operaciones determina un nodo virtual más lento, a una velocidad real que depende de la plataforma experimental que se utilice.

[1]`NOP` en la plataforma x86.

2.1.4 Instalación automatizada de plataformas distribuidas

Algunos sistemas distribuidos tienen un costo administrativo mayor que las aplicaciones MPI, requiriendo, por caso, la creación y propagación de archivos de configuración, generados a medida de la plataforma virtual, o la inicialización de servicios determinados.

Para poder solucionar estos problemas desde la propia herramienta, incorporamos a SherlockFog la capacidad de copiar archivos de configuración y ejecutar comandos en cada nodo virtual por medio de una interfaz de red administrativa adicional que no tenga en cuenta la topología emulada.

El uso de esta característica provee a la red virtual de un mecanismo de "escape" hacia la red física, por lo que se debe tomar la precaución de que el servicio a ser evaluado no reciba conexiones sobre esa red. Las interfaces adicionales pueden ser fácilmente apagadas o prendidas por el usuario según lo requiera, y así evitar este tipo de inconvenientes.

3 Experimentos sobre los NAS Parallel Benchmarks

En la sección 1.1 hemos introducido tres familias de metodologías para el estudio de los sistemas distribuidos: emulación, simulación y experimentación. Cada una propone un conjunto de herramientas, algunas de propósito general, y otras de propósito específico.

En el paradigma de emulación, existen diversas herramientas que hacen uso de la funcionalidad de *traffic shaping* existente en los sistemas operativos modernos para emular una red y ejecutar aplicaciones distribuidas sobre la misma, pero ninguna se focaliza específicamente en MPI sobre entornos heterogéneos.

Por el lado de la simulación, podemos mencionar que este enfoque permite el usuario explorar escenarios que son difíciles de configurar en ambientes reales. En este caso, la aplicación es ejecutada en un entorno completamente simulado, requiriendo usualmente que el usuario modifique o recree la aplicación a ser evaluada de una manera que pueda ser utilizada por el simulador.

En este capítulo, exploramos el uso de SherlockFog en el estudio de la *performance* de aplicaciones MPI en entornos de red heterogéneos. Proponemos experimentos de validación para la metodología sobre una aplicación con un patrón de comunicación conocido para mostrar que nuestra herramienta permite reproducir diferentes escenarios de red fielmente. Finalmente, evaluamos algunos de los *benchmarks* de NPB-MPI (ver sección 1.4) como ejemplos representativos de código científico que es implementado utilizando el modelo de programación de MPI, a los efectos de estudiar la incidencia de la latencia en un topología de red en particular.

3.1 Validación

Para nuestros experimentos de validación definimos la topología isles, que representa dos *clusters* de recursos computacionales interconectados a través de un único enlace distinguido. La latencia de este enlace indica la distancia en términos de tiempo de comunicación. Este escenario representa dos conjuntos de nodos en el borde de la red que están conectados a una infraestructura común, como puede ser Internet.

Sea n el tamaño de la red, las reglas de ordenamiento de procesos son las siguientes:

1. El enlace distinguido conecta el primer nodo (nodo 0) con el último (nodo $n-1$).
2. Los nodos son particionados mitad y mitad entre ambos *clusters*.
    - Los nodos 0 a $\lfloor \frac{n-1}{2} \rfloor$ se asignan al primer cluster.
    - Los nodos $\lceil \frac{n-1}{2} \rceil$ a $n-1$ se asignan al segundo cluster.

3. Los nodos conectados por el enlace distinguido se configuran como los “nodos de salida” de cada cluster.

4. El resto de los nodos de cada cluster se conecta únicamente al nodo de salida que le corresponde.

Adicionalmente, utilizamos la topología barabasi, un grafo aleatorio generado utilizando el modelo Barábasi-Albert para redes libres de scala con conexión preferencial. Este modelo de conectividad es común en redes como Internet [20]. El valor utilizado para el parámetro del modelo es $m_0 = 2$. En este caso, la asignación de procesos es completamente aleatoria. La latencia de todos los enlaces es uniforme, con lo que el tiempo de comunicación estará determinado por la cantidad de pasos (hops) que deban realizarse.

En las siguientes secciones, mostramos que SherlockFog puede emular diferentes condiciones de red por medio de la comparación entre la predicción teórica y los resultados obtenidos.

3.1.1 Emulación de la latencia de los enlaces

Para mostrar cómo funciona la emulación de la latencia de los enlaces, necesitamos una aplicación cuyo patrón de tráfico sea tal que se pueda obtener una expresión analítica para el tiempo total de comunicación. De esta manera, podemos comparar el resultado teórico contra la salida de la herramienta.

En particular, utilizamos una implementación de un *token ring*. Cada nodo conoce sus vecinos y el número de orden dentro del anillo. El tamaño del *token* fue configurado en un entero (4 bytes). La cantidad de veces que el token es recibido por el iniciador (cantidad de rondas) es parámetro de la aplicación.

Analizamos el número total de mensajes en la red y el tiempo de ejecución en las siguientes dos implementaciones de este problema:

- **Token Ring**: implementación utilizando sockets TCP. El uso de esta versión se fundamenta en poder tener un control más fino del protocolo utilizado.

  Cada nodo inicia manualmente su propio proceso con los siguientes argumentos de línea de comandos:

  - Número de rondas: cuántas veces debe volver el *token* al primer proceso.
  - Posición en el anillo: número de proceso, similar al *rank* en MPI.
  - Lista de nodos (ordenada): los nombres de host de cada nodo que conforma el anillo.

- **MPI Token Ring**: la misma aplicación, pero utilizando MPI para la comunicación.

Debido a que el patrón de tráfico es conocido, si mantenemos la topología igual, pero incrementamos la latencia de uno o más enlaces, es sencillo estimar cuánto más tardaría una aplicación en completar su ejecución con respecto a las condiciones de red originales. El incremento se calcula de la siguiente manera: sea N el número de nodos en la topología, t_0 el tiempo original de ejecución, $c_{i,j}$ el número total de envíos que se realizan del nodo i al nodo j y $w_{i,j}$ el peso del camino mínimo (medido en la demora mínima para atravesar ese camino) del nodo i al nodo j, el tiempo esperado de ejecución t_e se define como:

$$t_e = t_0 + \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} c_{i,j} \cdot w_{i,j} \tag{3.1}$$

Cabe mencionar que la ecuación 3.1 representa el tiempo esperado de ejecución de manera precisa únicamente debido a que el patrón de tráfico de las aplicaciones de prueba es completamente secuencial. De otra manera, habría que considerar el solapamiento entre cómputo y comunicación, que en este caso es nulo.

Calculamos los tiempos esperados teóricos y los obtenidos al modelar la topología con SherlockFog para diferentes valores de latencia en ambas familias de topologías de red, y para distintos valores de número total de nodos. Los resultados muestran que el error predicho difiere del tiempo medido en menos de 1% en todos los casos, tanto en el caso de **Token Ring** como de **Token Ring MPI**.

Podemos concluir que la latencia es modelada de manera precisa por nuestra herramienta al ejecutar sobre redes emuladas aplicaciones que utilizan MPI para la comunicación.

3.2 Resultados

En esta sección, mostramos el efecto de la latencia en diferentes escenarios de la versión MPI de los NAS Parallel Benchmarks [21]. Los mismos fueron descriptos en la sección 1.4. Hemos elegido tres de los kernels (IS, CG y MG) y dos de las seudoaplicaciones (BT y LU), y evaluamos la pérdida de *performance* en las topologías de red de la familia isles. Todos los *benchmarks* fueron ejecutados utilizando SherlockFog para modelar la topología, incrementando en cada experimento la latencia del enlace distinguido hasta 100 veces para tres clases distintas de tamaño de problema (A, B y C). Los experimentos fueron repetidos 5 veces.

Nuestro interés yace en entender cómo la *performance* de estos *benchmarks* varía con respecto a no incrementar la latencia en la misma topología.

Los resultados de esta sección describen el incremento del tiempo total de ejecución como función del incremento en la latencia para cada tamaño de red. Las áreas semi-transparentes sobre las curvas muestran el desvío estándar de cada serie de datos. Mostramos algunos resultados representativos en la figura 3.1.

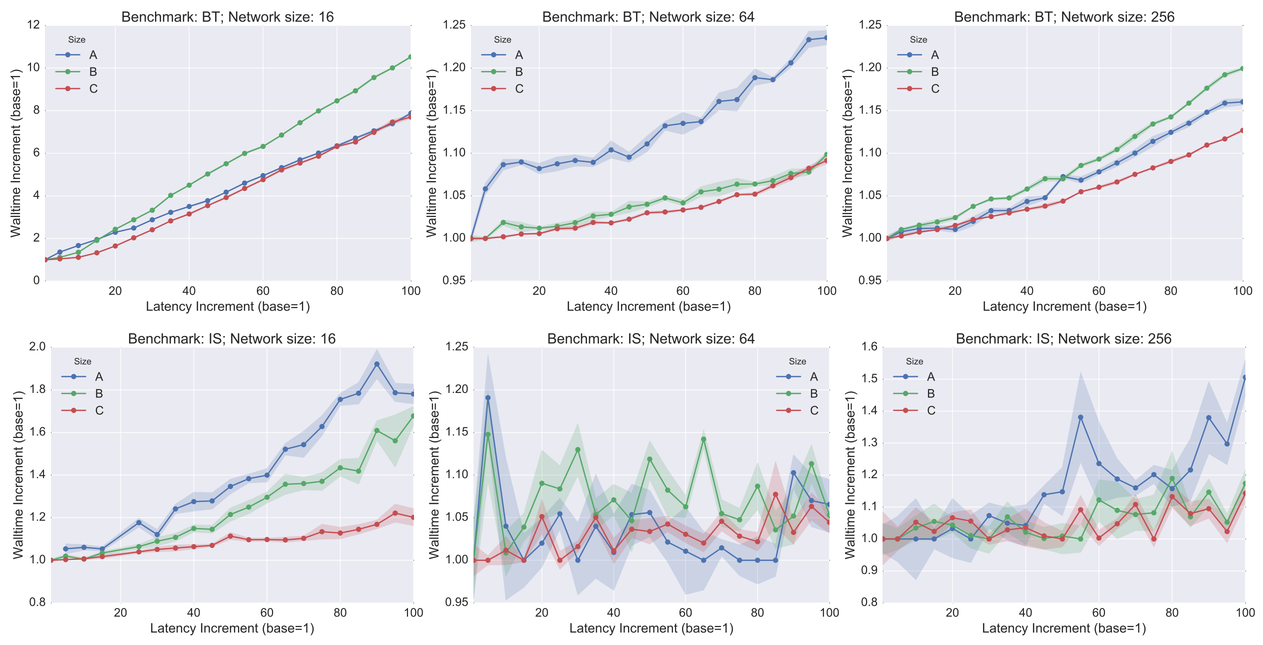


Figure 3.1: Incremento del tiempo total de ejecución en las topologías isles como función del incremento en la latencia del enlace distinguido en NAS Parallel Benchmarks.

3.3 Conclusiones

En este capítulo, validamos la capacidad de modelar topologías con enlaces de distintas latencias en SherlockFog y analizamos cinco *benchmarks* conocidos que utilizan MPI para resolver problemas del campo de CFD. Propusimos una topología de red que modela dos clusters interconectados, y mostramos el impacto del incremento de la latencia del enlace distinguido en la *performance* de cada aplicación.

Todos los resultados obtenidos muestran un impacto lineal o sublineal en esta topología en particular, abriendo las posibilidades de utilizar recursos computacionales distribuidos y cada vez más ubicuos.

4

SherlockFog en escenarios IoT

El paradigma de IoT define una visión de la computación en la cual la mayoría de los objetos que nos rodean van a estar conectados a algún tipo de red [22]. Dada esta organización, se generan grandes cantidades de datos que deben ser almacenados, procesados y presentados de manera eficiente y amigable al usuario. Este modelo está íntimamente ligado a otros paradigmas, tales como Cloud Computing y sus derivados como Fog y Edge Computing. IoT se construye sobre la proliferación de distintos tipos de sensores y sobre redes inalámbricas ya instaladas, sobre una visión que va más allá de la computación tradicional hacia la computación ubicua: los objetos del día a día están conectados entre sí y hacia otras infraestructuras de manera integrada, *desapareciendo* de la conciencia del usuario. Gubbi *et al.* proponen las siguientes demandas para alcanzar este fin:

1. Un entendimiento compartido de la situación de sus usuarios y dispositivos.
2. Arquitecturas de software y redes ubicas de comunicación para procesar y transmitir la información contextual.
3. El uso de técnicas analíticas que apuntan a un comportamiento autónomo e inteligente.

Al incrementarse las capacidades de los dispositivos IoT, nuevas aplicaciones aparecen en escena que pueden beneficiarse de este paradigma. Un enfoque preexistente para utilizar recursos distribuidos para cómputo es el de *volunteer computing* [23]. En dicho modelo, los participantes ceden poder de cómputo en desuso para procesar parte de un problema que no puede ser solucionado individualmente de manera sencilla. Este enfoque funciona para problemas trivialmente paralelizables [24]. Dicho éxito plantea la pregunta de si es posible el uso de dispositivos IoT para cómputo paralelo.

Este capítulo plantea una respuesta a esta pregunta utilizando SherlockFog para modelar una plataforma IoT en la que aplicaciones MPI puedan ser ejecutadas.

4.1 IoT-LAB

La plataforma FIT IoT-LAB [25] consiste de muchos dispositivos IoT localizados en sitios geográficamente distribuidos y con acceso unificado. Los nodos se encuentran en diversas ciudades de Francia. La plataforma permite el acceso *bare-metal* a nodos IoT de diferentes tipos y poder de procesamiento.

4.1.1 Uso de la plataforma

La plataforma provee una interfaz web para administración. Un usuario crea una cuenta en el portal, y la interfaz le permite cargar ejecutables en los dispositivos y acceder directamente a los *gateways* de los nodos.

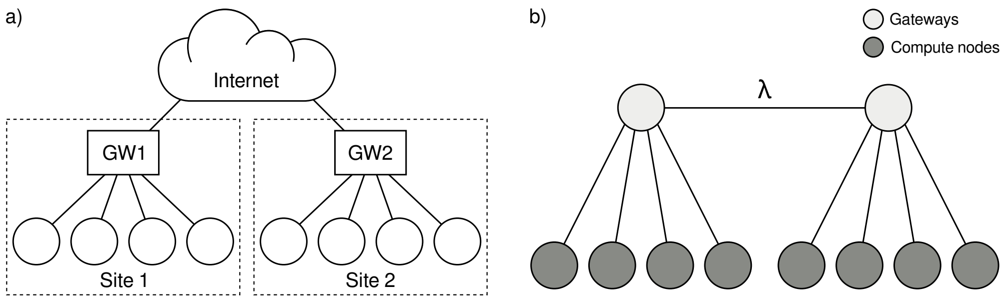


Figure 4.1: Topologías de FIT IoT-LAB: a) Física and b) Emulada

Además, existe un mecanismo adicional para acceder a los recursos por medio de una interfaz de línea de comandos. Utilizando cualquiera de las interfaces, el usuario puede hacer una reserva de un número arbitrario de nodos de diferentes tipos en uno o más sitios, y correr los experimentos sobre los mismos.

Para verificar el estado de los nodos, existen herramientas de monitoreo que informan datos sobre la red, el consumo energético y otros sensores.

Una vez concretada la reserva, la lista de nodos puede accederse por medio del comando `experiment-cli`. En los nodos que así lo permiten, una vez que los mismos están completamente inicializados, se permite el acceso como superusuario por Secure Shell (SSH) desde el nodo *gateway* o desde cualquier parte de Internet (por medio del protocolo IPv6).

En este capítulo utilizamos únicamente los nodos A8, que son los más poderosos de la plataforma, y los únicos que son capaces de instanciar una instalación completa de Linux.

4.2 Validación

4.2.1 Escenarios

Proponemos un escenario que puede ser reproducido en IoT-LAB para validar el mecanismo de simulación de CPUs más lentas que implementamos en SherlockFog.

Cada experimento utiliza un par de nodos A8 de la plataforma. Entre ambos escenarios, la diferencia radica en cuáles son los sitios a los que pertecene cada nodo del par:

Mismo sitio Ambos nodos se encuentran en el mismo sitio físico (Saclay).

Dos sitios Cada nodo se encuentra en un sitio diferente (Saclay y Grenoble).

Este escenario plantea una dificultad ya que IoT-LAB no permite conexiones entre nodos en distintos sitios sino a través de IPv6, que desafortunadamente ninguna implementación de MPI soporta. Por este motivo, extendimos MPICH para que soporte esta familia de direcciones. La topología real se muestra en la figura 4.1.

La extensión involucró los siguientes cambios:

- Utilizar sockets IPv6 para toda comunicación.
- Utilizar API con soporte de IPv6 en el código de resolución de nombres de *host*.
- Modificar los *parsers* de parámetros y de archivos de *hosts* para aceptar literales de IPv6, según el formato definido por el RFC 2732 [26].
- Utilizar para IPv4 direcciones *mapeadas* a IPv6, permitiendo el funcionamiento de nodos con soporte para cualquiera de las dos familias. Este *mapeo* está definido en el RFC 4291 [27].

Se midieron las latencias entre sitios y en el mismo sitio en IoT-LAB utilizando ICMP, que arrojó los siguientes valores:

Mismo sitio 0.3 ms de latencia.

Dos sitios 5 ms de latencia.

Estos parámetros fueron utilizados para configurar SherlockFog de modo que se corresponda la topología lógica inducida cuando se utilizan solamente dos nodos. La latencia del enlace distinguido (mostrado en la figura como enlace λ) se configuró al valor de **Dos sitios**, mientras que todo el resto de los enlaces fueron configurados con el valor de **Mismo sitio**.

La metodología propuesta es la siguiente:

1. Elegir una aplicación y parámetros de ejecución.
2. IoT platform $\rightarrow$ Ejecutarla en un único nodo y medir el tiempo total.
3. IoT platform $\rightarrow$ Ejecutarla en dos nodos (**Mismo sitio**) y medir el tiempo total.
4. IoT platform $\rightarrow$ Ejecutarla en dos nodos (**Dos sitios**) y medir el tiempo total.
5. Experimental platform $\rightarrow$ Executarla en un único nodo:
    - Utilizar Pin para inyectar no-operaciones.
    - Ejecutarla en esta configuración y medir el tiempo total.
    - Repetir hasta encontrar el número de no-operaciones que determina un tiempo de ejecución similar al de la plataforma IoT.
6. Experimental platform $\rightarrow$ Configurar el mismo número de no-operaciones encontrado en el paso anterior para reproducir el experimento en dos nodos en SherlockFog.
7. Experimental platform $\rightarrow$ Repetir la corrida anterior en SherlockFog utilizando Pin para inyectar no-operaciones mientras se varían los parámetros de latencia para que se corresponda a ambos escenarios IoT.
8. Comparar el tiempo total de ejecución en ambos escenarios IoT con los valores predichos por SherlockFog.

Esperamos que la inyección de no-operaciones medida en un único nodo (en el cual no hay comunicación de red) funcione como un buen estimador para esa aplicación utilizando los mismos parámetros de entrada.

Utilizamos los NPB-MPI CG (clases S y A) y MG (clase S) para evaluar el funcionamiento de nuestra propuesta. MG clase A fue eliminado del conjunto de prueba, ya que se por limitación de memoria no era posible ejecutarlo en los nodos de IoT-LAB.

4.2.2 Resultados

Los resultados obtenidos muestran que para NPB-CG clase A, la sobreestimación que realiza nuestra técnica es del orden del 10% con respecto a la plataforma real, si bien dicho error se mantiene consistente incluso al incrementar la latencia del enlace distinguido. Esto representa una variación de 4 ó 5 segundos en una corrida de 50 segundos de duración.

En el caso de NPB-CG clase S, la predicción se corresponde con la plataforma experimental dentro del error experimental, considerando que el tiempo total en la configuración **Mismo sitio** muestra una variabilidad mucho mayor en SherlockFog.

En el caso de NPB-MG clase S, la subestimación es de casi el 20%. Los datos muestran que al ver el tiempo total de ejecución, la diferencia es de solo 0.2 segundos en el tiempo total. Esto se debe a que el tiempo total de ejecución es muy pequeño. Sin embargo, incluso en este caso, la tasa de error se mantiene consistente al incrementar la latencia del enlace distinguido.

Estos resultados muestran que, si bien la técnica no permite reproducir la *performance* del sistema más lento con una fidelidad plena, el método proporciona un buen estimador de poder de cómputo que puede ser utilizado en SherlockFog para analizar los efectos que provocan cambios en la topología en entornos heterogéneos.

4.3 Experimentos

En esta sección, proponemos un conjunto de experimentos para evaluar, utilizando SherlockFog, los *benchmarks* NPB-CG y NPB-MG en escenarios IoT con poder de cómputo heterogéneo. Primero, modelamos una topología de red en la cual 16 nodos son distribuidos equitativamente en dos sitios, siguiendo la topología mostrada en la sección anterior. Para modelar esta topología en SherlockFog, utilizamos dos nodos adicionales que funcionan como *gateways* entre los dos sitios y que no realizan ningún cómputo. Una versión escalada de la topología resultante (6 nodos, 3 nodos por sitio) se puede ver en la figura 4.1. Del lado a), la topología real de IoT-LAB. Del lado b), la numeración de nodos utilizada en SherlockFog para modelar dicha topología. Los nodos extra utilizados solamente para comunicación se muestran en gris claro.

4.3.1 Escenarios

Cada escenario consiste en configurar el poder de cómputo de cada nodo en un valor fijo. Utilizamos dos valores diferentes según la categoría: *lento* y *rápido*.

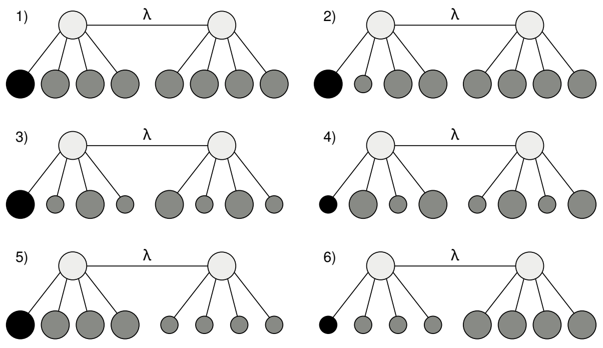


Figure 4.2: Descripción de los escenarios experimentales.

Los mismos se muestran esquemáticamente en la figura 4.2. Los círculos grandes muestran los nodos rápidos, mientras que los más pequeños son los lentos. El nodo principal se marca en negro. Los nodos en gris claro se utilizan solo para comunicación.

La descripción completa es la siguiente:

(1) Todos los nodos son rápidos.

(2) Solo un nodo, no principal y localizado en el primer sitio, es lento.

(3) Nodos vecinos según el número de proceso de MPI son alternativamente rápidos o lentos.

(4) Nodos vecinos según el número de proceso de MPI son alternativamente lentos o rápidos.

(5) Todos los nodos en el primer sitio son rápidos, mientras que el resto es lento.

(6) Todos los nodos en el primer sitio son lentos, mientras que el resto es rápido.

Ambos *benchmarks* fueron ejecutados en SherlockFog utilizando diferentes valores de latencia para el enlace distinguido, con valores de 0.3 a 5 ms en cada escenario.

Mostramos los resultados para NPB-CG en la figura 4.3. El escenario 1) es utilizado como base para mostrar la pérdida de *performance* con respecto a otros escenarios heterogéneos. En dicha configuración base, se puede observar que la latencia afecta el tiempo total de ejecución linealmente.

El escenario 2) muestra que si se enlentece solamente un nodo en el primer sitio, esto resulta en que el *benchmark* tarda cerca de un 60% de tiempo más en finalizar. Además, el efecto de la latencia se reduce significativamente. En este caso, la diferencia en la *performance* reduce la incidencia de la latencia mayor.

Los escenarios 5) y 6) muestran resultados similares al 2), si bien el tiempo total de ejecución es mayor.

Los escenarios 3) y 4), sin embargo, muestran un tiempo total de ejecución diferente a los escenarios 5) y 6), si bien la pendiente es similar a la comparación base. Alternar nodos vecinos rápidos y lentos—un nodo interno está conectado a los que tienen número de proceso inmediato anterior e inmediato posterior—muestra una *performance* similar a un escenario en el que todos los nodos son lentos.

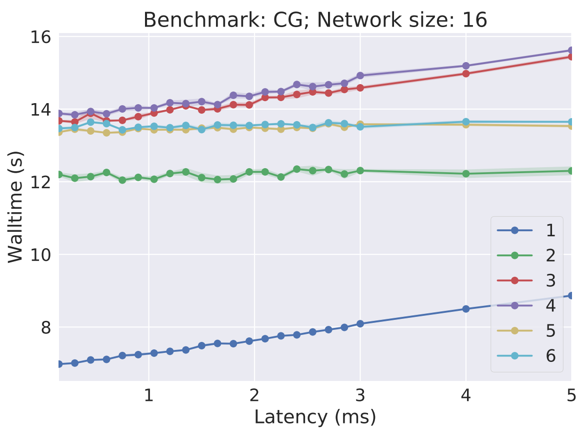


Figure 4.3: Tiempo total de ejecución como función de los incrementos en la latencia en un escenario con dos sitios; NPB-CG sobre SherlockFog.

5

Hacia una plataforma para el estudio de sistemas basados en blockchain

Hasta el momento, nos hemos focalizado en el estudio de aplicaciones de cómputo científico, particularmente aquellas que están implementadas utilizando MPI. Sin embargo, SherlockFog fue concebido como una herramienta de propósito general y creemos que las técnicas de emulación de redes representan un enfoque viable para estudiar otro tipo de sistemas distribuidos.

Una familia que ha generado recientemente un gran interés es la de los sistemas basados en *blockchain*. La aplicación más común de este tipo de tecnologías es la implementación de criptomonedas descentralizadas y distribuidas. Crear una criptomoneda exitosa requiere un entendimiento adecuado de los algoritmos de consenso, la estructura de la red y los efectos que las reglas definidas tienen sobre la estabilidad y la escalabilidad del sistema. Este estudio dista de ser trivial. Además, cada cambio propuesto a monedas actualmente en funcionamiento generan importantes debates en la comunidad, ocasionando en algunos casos situaciones denominadas *hard forks*[1] debido a la incapacidad de alcanzar el consenso entre los participantes. Claramente, bajo estas circustancias, es importante identificar algún mecanismo para estudiar objetivamente cambios en los parámetros de la red y el agregado de nuevas funcionalidades a las implementaciones existentes, con el objetivo de estudiar su pertinencia.

En este capítulo, utilizamos SherlockFog para proponer una metodología para el estudio del protocolo de la criptomoneda Ethereum, como ejemplo representativo de las criptomonedas basadas en *blockchain* que utilizan Proof of Work (PoW), al mismo tiempo que proveemos un mecanismo para reducir los requerimientos de hardware de esta plataforma experimental, pero sin dañar su fidelidad.

5.1 Cambios al cliente de referencia de Ethereum

Utilizamos el cliente `geth` de Ethereum como implementación de referencia para mostrar el potencial de nuestra metodología. El proceso de minado fue modificado para producir un patrón de geneneración de bloques similar al real, pero sin requerir un gran poder de cómputo. Dicho cliente, además, fue instrumentado para almacenar eventos de red en cada nodo para su posterior procesamiento. Los mensajes instrumentados fueron `Status`, `NewBlockMessage`, `NewBlockHashes`, `GetBlockHeaders`, `BlockHeaders`, `GetBlockBodies` y `BlockBodies`.

[1] Particionamiento de la red en dos o más subconjuntos de nodos con vistas de la *blockchain* no consensuadas.

5.1.1 Minado simulado

En sistemas basados en PoW, como es el caso de Ethereum, los mineros utilizando su propio poder de cómputo para encontrar nuevos bloques candidatos a ser incluidos en la *blockchain*. Si quisiéramos emular la red completa, de miles de nodos, esto requeriría cantidades inalcanzables de poder de cómputo en los nodos de la plataforma experimental. Las capacidades de emulación del sistema también son limitadas por el poder de *hashing* (cálculo de *hashes* criptográficos por unidad de tiempo) máximo disponible.

Nuestra solución propone considerar las características estadísticas del proceso de minado, y utilizar dicha información para generar una entrada en el sistema que no requiera poder de procesamiento. En sistemas PoW, la dificultad de la red es ajustada dinámicamente de manera que el poder de *hashing* combinado de cada minero en la red encuentra un nuevo bloque con un tiempo medio de t unidades de tiempo, describiendo un proceso de Poisson.

Debido a que este minado simulado no genera bloques válidos, esto requiere modificar el proceso de validación de los nodos para que los mismos sean aceptados. Este cambio requiere considerar el tiempo que demora dicho proceso en el minado normal, ya que no es despreciable. Nuestra implementación simula todo el proceso realizando una espera sin cómputo, en lugar de utilizar el algoritmo de fuerza fruta que lleva a la obtención de un nuevo bloque.

5.1.2 Instrumentación de eventos

Modificamos el cliente para que cada nodo genere un archivo de *log* en el que los eventos de red sean grabados, con toda la información pertinente, y con un *timestamp* (marca temporal) con el que se los pueda ordenar temporalmente.

Para asegurar que los eventos de todos los *logs* combinados puedan ser procesados en conjunto, los relojes de cada nodo de la plataforma experimental deben estar sincronizados. Utilizamos clientes NTP sobre la red no emulada para este fin. Evaluamos la precisión de este mecanismo, obteniendo que el número de eventos almacenados en un *log* cuyo *timestamp* de generación sea posterior al evento en el que dicho mensaje es recibido no es nunca mayor al 0.1%, incluso en las condiciones de red menos favorables.

5.2 Métricas de *forks*

Construir una *blockchain* se trata de establecer consenso de manera descentralizada. Idealmente, un protocolo de consenso debería resolver todos los conflictos entre bloques en competencia, al mismo tiempo que se le permite a cada nodo ser informado del estado real. Una situación a evitar es la de los *forks*: dos o más nodos que tienen *vistas* diferentes de la *blockchain*. Varias características de la red pueden tener incidencia sobre la generación de *forks*, tales como los tiempos de generación de nuevos bloques y la topología de la red.

Definimos tres métricas para estudiar la *performance* de nuestra red privada con respecto a la generación de *forks*.

5.2.1 Proporción de bloques huérfanos

Esta proporción se define contando la cantidad de bloques que fueron minados por cualquiera de los mineros, pero que finalmente no forman parte de la *blockchain*. Dicha cantidad es normalizada por el número total de bloques minados. Esta métrica nos ofrece una perspectiva sobre cuánto cómputo fue desperdiciado en la red para producir la *blockchain* final.

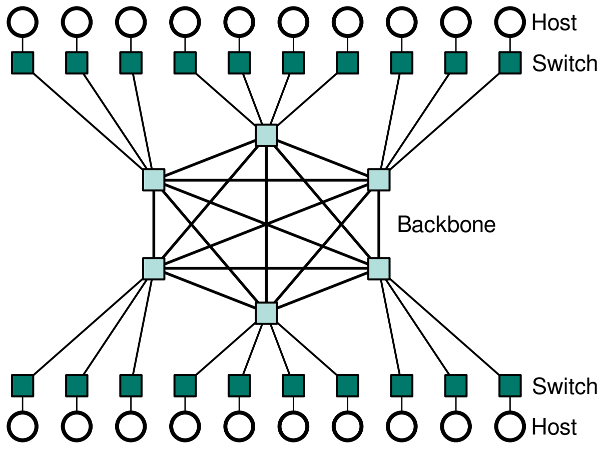


Figure 5.1: Topología World

5.2.2 Bloques apuntando a la cadena principal

Esta métrica se define contando la cantidad de bloques que son apuntados por uno o más bloques, normalizado por la cantidad total de bloques de la cadena principal. Si la red fuera completamente libre de *forks*, todos los bloques deberían ser apuntados por 0 ó 1 bloques, resultando en un valor de 0 para esta métrica. En otro caso, su valor se incrementaría conforme la cantidad de ramas de la cadena principal sube, sin importar la profundidad que alcancen las mismas.

5.2.3 Desvío con respecto al tiempo de *target*

Una *blockchain* en correcto funcionamiento debería producir un nuevo bloque cada *target* unidades de tiempo en promedio. Sin embargo, si algunos bloques no terminan formando parte de la cadena principal debido a *forks*, esto implica que la cantidad de tiempo que se requiere para que un bloque efectivamente alcance la cadena principal será necesariamente mayor que el *target*, ya que un número mayor de bloques debe ser minado. Esta métrica representa dicha noción. Se calcula dividiendo todos los tiempos de generación de bloques agregados por el número de bloques que terminaron en la cadena principal, y normalizando dicho valor al tiempo de *target* configurado en el sistema.

5.3 Topologías de red

Debemos distinguir dos tipos de topologías: la física y la lógica. La topología física se refiere a la red subyacente sobre la cual el tráfico es *ruteado*. La lógica, por otra parte, se relaciona con el patrón de comunicación que sigue la aplicación.

Dos tipos de topología física fueron definidos para nuestros experimentos: *World* y *Fully Connected*.

5.3.1 World

En esta topología, cada nodo tiene una etiqueta que la indica de qué país es. Cada nodo se conecta a un único switch, que a su vez se conecta al *backbone* de la red. El *backbone* es una clique de switches, cada uno de los cuales representa el nodo de salida de un país en particular. La estrategia de *ruteo* utilizada es tal que si dos nodos que se encuentran etiquetados en el mismo país se desean comunicar, utilizarán el nodo de salida de su país para conectarse, mientras que dos nodos de distintos países utilizarán el enlace que conecta el nodo de salida del país del primero con el del segundo.

Mostramos esto esquemáticamente en la figura 5.1.

La cantidad de nodos asignados a cada país es proporcional a la cantidad de clientes de Ethereum que hay en dicho país, según la estadística publicada por `ethernodes.org` [28].

El modelo de latencia fue generado utilizando información de latencias promedio entre países. La latencia de un enlace que conecta dos switches del *backbone* fue definida como la mitad de la latencia promedio que existe entre esos países. El enlace que conecta un switch interno de un país con su correspondiente *backbone* fue definido como la mitad de la latencia intra-país indicada en los datos utilizados. Este modelo tiene limitaciones ya que algunos países que tienen el mayor número de nodos también presentan una enorme variación en sus latencias intra-país, debido a la extensión geográfica de los mismos (por ejemplo, China, Rusia, Estados Unidos). Sin embargo, dicho modelo puede ser mejorado utilizando información más granular sobre la distribución de latencias de la red.

5.3.2 Fully Connected

Esta topología es un grafo K_N, en el que cada enlace tiene la misma latencia (1 ms).

La estrategia de ruteo en este caso es tal que cada nodo es alcanzable en un único salto (utilizando el enlace que lo conecta directamente con el destino).

5.3.3 Topología lógica

La topología lógica refiere a las conexiones que establece un cliente con otro u otros en la red.

La red lógica fue definida de manera aleatoria para cada topología y tamaño de red, configurando los clientes estáticamente con una lista de vecinos de 2 ó 3 nodos elegidos aleatoriamente. El descubrimiento de nodos nuevos fue desactivado de todas las pruebas.

En todas las configuraciones utilizadas, dada una red de N nodos, se eligieron $\frac{N}{2}$ nodos como mineros de manera aleatoria.

5.4 Validación

Para validar nuestra metodología, planteamos un experimento en el que comparamos los eventos de generación de bloques de una red con minado simulado sobre SherlockFog contra un escenario real en una red privada de Ethereum.

El escenario real elegido fue un laboratorio de computadoras de 20 nodos conectados a una red Ethernet, donde cada nodo corre nuestro cliente instrumentado. Los clientes fueron inicializados con el minado simulado desactivado y un tiempo de *target* prefijado en 21 segundos. La mitad de nodos fue configurada para minar bloques, utilizando para dicha operación un único core en CPU.

En el caso del escenario sobre SherlockFog, la topología física utilizada es un grafo K_{20} (Fully Connected). La espera de cada enlace fue configurada en 0.1 ms, respetando los valores medidos en el escenario real. La misma topología lógica aleatoria fue elegida en ambos casos.

Sean t_n y t_{n+1} los puntos en el tiempo en los cuales los bloques n y $n+1$ fueron minados, almacenamos las demoras en la generación de un bloque $t_{n+1} - t_n$ de los primeros 1000 bloques that efectivamente alcanzaron la cadena principal en ambos escenarios.

Los resultados obtenidos se pueden ver en la figura 5.2, y muestran que la demora en la generación de bloques es similar en ambos escenarios, y que siguen una distribución exponencial con media similar. Además, la proporción de *forks* es similar en ambos casos.

Podemos concluir, de esta manera, que el minado simulado en esta topología se comporta de manera estadísticamente similar al sistema real en una red local.

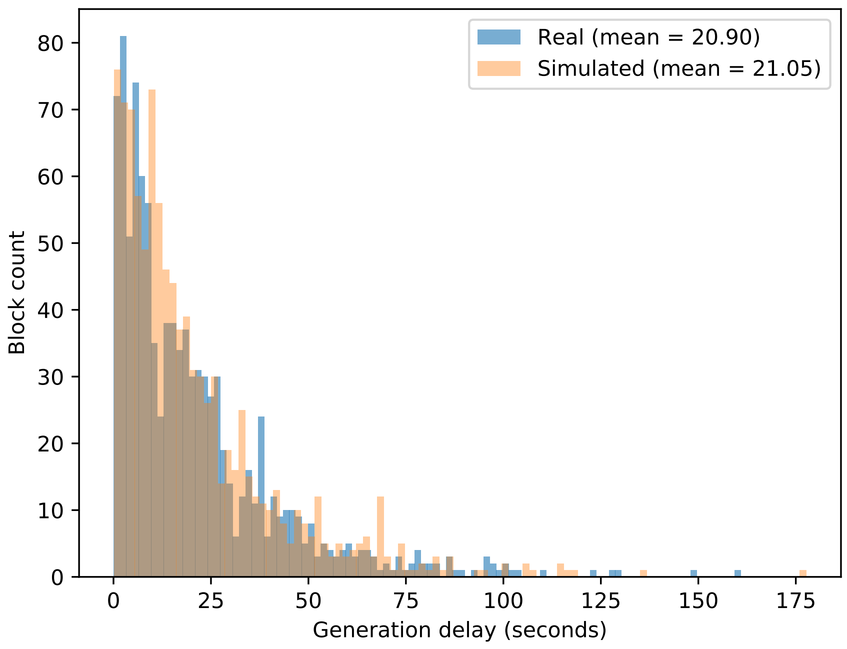


Figure 5.2: Distribución de los tiempos de generación de bloques

5.5 Resultados

Analizamos las tres métricas definidas para las dos topologías World (50, 100 y 200 nodos) y Fully Connected (50 y 100 nodos). Los tiempos de *target* utilizados varían entre 0.1 y 20 segundos. La plataforma experimental consistió de 6 computadoras de escritorio con procesadores Intel Core i7-2600.

Presentamos un resumen de los resultados importantes:

- **Proporción de bloques huérfanos**: La proporción se reduce conforme crece el tiempo de *target*. Este comportamiento es similar para todos los tamaños de red y en ambas topologías. Los tiempos de *target* menores a 1 segundo muestran ser demasiado cortos para ambas topologías.

- **Bloques apuntando a la cadena principal**: Esta proporción también se reduce conforme se incrementa el tiempo de *target*. Un caso interesante es el de *target* 0.1 segundos, donde el valor obtenido se incrementa más lentamente (o decrementa en el caso de World, 50 nodos) con respecto al valor inmediato superior. Este efecto se debe a la ramificación de la *blockchain* que ocurre en ramas que ya forman parte de un *fork*, por lo que en esta métrica no son contados.

- **Desvío con respecto al tiempo de *target***: el resultado más interesante de esta métrica es la observación de que los tiempos de *target* menores requieren hasta 10 veces más tiempo para producir un bloque en la cadena principal. El desvío decrece exponencialmente a medida que el tiempo de *target* se acerca a 1 ó 2 segundos (dependiendo del tamaño de la red), convergiendo lentamente al tiempo configurado a medida que se sigue incrementando. También es interesante notar que se requiere un *target* de 10 segundos o más para que el desvío alcance valores cercanos a 1.

6
Conclusiones y trabajo futuro

El estudio de los sistemas distribuidos propone retos que se derivan de la dificultad que conlleva construir entornos realistas en los cuales los efectos de distintas condiciones de red y plataformas heterogéneas puedan ser modelados. Esto impone la necesidad de herramientas específicas que ayuden a los desarrolladores e investigadorse a comprender acabadamente la indidencia del entorno en la *performance* del sistema. En este trabajo presentamos SherlockFog, una herramienta de emulación de redes que se focaliza en el estudio de aplicaciones distribuidas en entornos heterogéneos utilizando técnicas de virtualización livianas. Fueron estudidas dos clases de aplicaciones: las de cómputo científico paralelo utilizando MPI y los sistemas distribuidos que implementan tecnologías basadas en *blockchain*.

Las contribuciones principales son las siguientes:

- El modelado de la latencia en un escenario de Fog o Edge Computing para estudiar la degradación de la *performance* en varias aplicaciones representativas del campo de CFD.
- El modelado de la *performance* de cómputo de nodos IoT, también utilizados en el contexto de la ejecución de aplicaciones MPI, mostrando que al tratarse de nodos mucho más lentos, son más resistentes a variaciones en la latencia.
- El modelado de criptomonedas basadas en PoW, tomando como ejemplo representativo el caso de Ethereum, y utilizando una técnica para simular el minado de manera realista, y de esta manera reducir el poder de cómputo necesario para poner en funcionamiento la plataforma experimental.

Hemos mostrado que SherlockFog es una plataforma que mejora el estudio de sistemas distribuidos y aplicaciones paralelas bajo diferentes condiciones de red, proveyendo un entorno controlado para hacer pruebas, proponer experimentos e implementar cambios sobre los mismos.

6.1 Trabajo Futuro

A partir de nuestro trabajo se desprenden varias preguntas abiertas para continuar explorando el estudio tanto de las aplicaciones paralelas como los sistemas distribuidos en general.

En cuanto a las aplicaciones paralelas, algunos temas no explorados son los siguientes:

- Evaluar el *churn* (cambios en la conformación de la red por nodos que se desconectan o vuelven) en aplicaciones MPI, introduciendo cambios a implementaciones de las bibliotecas.

- Introducir mecanismos de resistencia a la pérdida de paquetes en las bibliotecas de MPI y evaluar su uso.
- Estudiar condiciones de red dinámicas, tales como cambios en tiempo de ejecución sobre la latencia, el ancho de banda y la pérdida de paquetes de uno o más enlaces.
- Extender el estudio de otras tecnologías que permitan implementar aplicaciones de cómputo paralelo.
- Emular enlaces de red inalámbricos para mejorar la precisión de los escenarios de emulación de IoT.
- Incorporar heurísticas en la implementación de los mensajes *collective* de MPI que permitan aprovechar información topológica para mejorar la ejecución en entornos de Fog/Edge Computing.

La aplicación de SherlockFog al estudio de sistemas basados en criptomonedas también deja varias preguntas abiertas, a saber:

- Evaluar límites de escalabilidad de los sistemas para modelar adecuadamente una red completa de una criptomoneda.
- Reproducir el mismo enfoque metodológico para estudiar otras criptomonedas basadas en PoW.
- Extender la plataforma para estudiar Proof of Stake u otros tipos de criptosistemas.
- Estudiar la selección de vecinos en Ethereum, con el objetivo de proponer estrategias novedosas que mejoren la escalabilidad de la red y reduzcan la concentración de la red.
- Modelar ataques a la red tales como cartel mining o selfish mining.

Part II

Thesis

1

Introduction

A *distributed system* is an abstract term to define a collection of independent entities that cooperate to solve a problem [29]. This has been applied to computing to characterize several types of systems that involve multiple components. Singhal and Shivaratari describe them as a collection of computers that do not share common memory or physical clock, and that communicate by message passing over a network. These computers are loosely coupled, behaving independently while they cooperate to address a problem collectively [30]. Tanenbaum uses this definition: A collection of independent computers that appear to the users of the system as a single coherent computer [31]. Goscinski uses this term to define a wide range of systems, from weakly coupled, such as Wide-area Networks (WANs), strongly coupled, such as Local-area Networks (LANs), to very strongly coupled, such as multiprocessor systems [32].

Distributed systems usually present the following features and characteristics:

1. *Inherently distributed computations*, such as those requiring consensus among autonomous parties.

2. *Resource sharing* of some components, such as peripherals and databases, that cannot be replicated in all sites due to cost or practicality concerns, but cannot be centralized as they would become a single point of failure for the system. Some distributed systems, such as the DB2 distributed database, propose clever architectures that provide a middle ground to these issues[1].

3. *Access to geographically distributed data and resources*: Communication networks allow resources to be accessed remotely where it is not feasible to replicate them.

4. *Enhanced reliability*: A distributed system can be designed to reduce single points of failure by replicating resources and executions, resulting in an increased reliability. This feature entails availability (resource should be accessible at all times), integrity (the data stored should be correct at all times) and fault-tolerance (the system should be able to recover from failures).

5. *Increased performance/Cost ratio*: Using geographically distributed data and resources results in an increased performance/cost ratio in several types of problems.

6. *Scalability*: The communication load may be distributed in a WAN, thus adding more processors does not necessarily pose a bottleneck for the communication network.

7. *Modularity*: Additional processors may be easily added or replaced.

[1]DB2 partitions data across several sites for performance concerns (no single site has to reply to all queries), while it also replicates some data for reliability (no single point of failure).

An interesting example of a distributed system that shows most of the previously described characteristics is Domain Name System (DNS) [33]. This system is used on the Internet to convert human-friendly hierarchical names to their corresponding network addresses. It is one of most massively distributed systems that is currently online and is one of the critical components of the Internet. The DNS service consists of multiple servers that are connected at different tiers to provide a distributed name database. The first tier is that of the *root servers*, which have replicated link information to the *top level domain* servers (6). Then, each top level domain server has a subdomain list with link information to which servers should be used to resolve those names (2), which should also have replicated information. Subdomains can be delegated from a higher domain by linking them with the appropriate entries (7) at a low administrative cost. Network addresses are resolved by querying one of the root servers, and following the link list to the domain of interest (3). For example, querying the network address for the `dc.uba.ar` name would involve the following steps:

i) Query one of the root servers and ask for the `ar` top level domain.
ii) Query one of the `ar` servers and ask for the `uba.ar` subdomain.
iii) Query one of the `uba.ar` servers and ask for the `dc.uba.ar` subdomain.
iv) Query one of the `dc.uba.ar` servers and ask for the `dc.uba.ar` name.

Responses can be additionally cached at local servers to reduce communication latency in frequent queries (4 and 5). Subdomain responses are called *authoritative*, whereas cached responses are *non-authoritative*, since they can be invalidated by the servers of that subdomain to preserve integrity. Subdomain delegation also allows decisions about which servers and names should be pointed to or resolved to be locally managed (1).

A related taxonomy is that of parallel systems, which refers to systems that are used to execute programs in parallel to solve a specific problem. We will call these programs *parallel applications*. Three types of parallel systems can be defined:

1. *A multiprocessor system* is a parallel system in which all processors have direct access to a shared memory region. This type of processors usually do not have a common clock. The memory architecture that is usually found on this type of systems is Uniform Memory Architecture (UMA), in which the access latency is the same no matter which memory address is being accessed. Processors have low communication latency to each other and are connected by an interconnection network. Interprocess communication is usually implemented by reading and writing from shared memory, although message passing is also used.

2. *A multicomputer parallel system* is a parallel system in which the processors do not have direct access to shared memory. The memory addresses of each processor may or may not form a common address space. A common clock is also not found on this type of systems. Processors have low communication latency and are connected by an interconnection network. The memory architecture in this type of systems is usually Non-Uniform Memory Architecture (NUMA) i.e. latency may vary from a shared location to another. This difference to UMA systems has to be considered when algorithms are designed for this type of systems. An example thereof are HPC clusters.

3. *Array processors* are very tightly coupled systems that possess a common clock but may or may not have a shared memory region. These are usually very specialized systems. Examples include DSP and image processors.

The primary use for parallel systems is to obtain a higher throughput by dividing the workload among different processors. This is not always feasible and depends deeply on the problem.

In general computing, the development process is facilitated by the use of different types of tools, such as specialized editors, debugging tools, profilers and testing frameworks [1]. These are usually executed in the developer's workstation. Distributed systems are no exception, but they propose additional challenges as the final execution environment extends over different networked systems, potentially covering multiple implementations of operating systems, underlying libraries, networking hardware, among others. Specific tools have been developed for these systems as well, such as parallel debuggers and profilers, in order to properly understand them and maximize their performance.

The size and complexity of distributed systems have been increasing over the last decades with the proliferation of heterogeneous technologies that are deployed on the Internet and other types of networks. Some examples include Internet services, ubiquitous computing environments, Cloud, IoT, storage and enterprise systems, sensor networks and mobile nodes [2]. As more and more vendors create cheaper and low-powered devices that can connect to the Internet or other types of networks, new types of services emerge that make use of this new infrastructure to provide distributed solutions to a variety of new problems. The complexity of distributed systems has been identified as an important problem, as many of the already deployed systems are very large and also difficult to manage, maintain and change, resulting in increased development costs for organizations. Therefore, building tools to understand such systems clearly becomes a necessity. Some challenges include being able to build a test deployment of a scale that is comparable to production and being able to single out which factors have an incidence on the performance of an application to properly identify and fix weaknesses. Differences in computational power or networking capabilities among nodes might have a direct effect on performance that is not clear without a proper testing framework that takes these characteristics into account.

This thesis focuses on the study of distributed and parallel applications in heterogeneous environments, taking into account both network connectivity and CPU heterogeneity. In order to accomplish this, we have developed a tool called SherlockFog, which is described in chapter 2, and propose a methodology to study distributed systems, focusing particularly on parallel applications that use the MPI library for scientific computing. Then, in chapter 3, we present several scenarios inspired by the Fog/Edge Computing paradigm to study how the performance is affected on several characteristic applications. Our objective is to analyze the feasibility of executing parallel applications using MPI on Fog/Edge scenarios. In chapter 4, we further extend the proposed platform to study parallel applications on IoT environments, improving SherlockFog with a mechanism to model a low-power CPU.

Finally, in chapter 5 we propose that this methodology could also be used to study distributed systems such as blockchain-based cryptocurrencies and show that SherlockFog can be used to analyze the Ethereum protocol in a controlled scenario. We conclude and provide pointers to future work in chapter 6.

1.1 Study of Distributed Systems

Distributed systems are pervasive in many areas of computing, ranging from scientific applications to content distribution systems. Many of these systems, such as P2P networks, can comprise millions of nodes, distributed all over the world. These are generally highly heterogeneous systems, in which many different subsystems and technologies interact simultaneously using common protocols. It has been a running effort for decades to assess the properties of these systems, such as reliability, resilience, performance or security. Most often, researchers rely on that for experimentation: they analyze the behavior by running the system under a particular scenario and capturing output data that could be of interest.

Our main focus is the study of the performance of this type of systems, which depends mainly on the communication protocol, the processing speed of individual components of the system (e.g. nodes), and the communication speed. In describing the communication performance of a distributed system, the two main metrics that define this feature are the *latency* and the *bandwidth* [11]. The latency is the time it takes for the first byte of a message to reach destination. In computer networks, it is also common to talk about link delay and round-trip time to define how long it takes for a message to reach destination and send an acknowledgment back respectively. The bandwidth is the rate at which the destination receives data after it has started to receive the first byte. This is limited by the maximum size of the message that can be "flying" at a given time (i.e. data that has already been sent but has not been received yet), which is a parameter of the interconnection technology.

Experimental work in distributed systems could be categorized in three different paradigms [3, 34, 4]: **Experimentation**, **Simulation** and **Emulation**. Each paradigm offers its own set of tools and methodologies. Most of these tools have evolved independently from each other, which makes it difficult to combine them for an augmented analysis. This work as a whole focuses on the **Emulation** paradigm, as we believe it is the most viable approach to study systems that are evolving or still under development.

The following sections define and characterize each approach.

1.1.1 Experimentation

Ideally, it would be desirable to test and evaluate distributed systems in an environment that resembles as much as possible its actual working conditions. If it were possible, obtaining experimental data would amount to running an instrumented version of the system that captures data of interest. However, large-scale and highly distributed systems are often unsuitable for this approach, as it requires access to geographically distributed nodes. Moreover, even if computational resources were available, testing on different scenarios under these circumstances can turn into an incredibly complex task. Many factors affect the performance of a distributed system that cannot be controlled "in the wild", deriving in irreproducible results. Examples include congestion levels in each physical link and individual system load. These factors could even depend on the time of the day in which the experiments are being made. Even so, this approach might be combined with statistical analysis to provide a realistic expectation of system metrics.

Planet-Lab [35] is a research network in which the experimentation paradigm can be used to develop and test distributed systems such as storage solutions, P2P systems, distributed hash tables and query processing frameworks on a global scale. It comprises of more than 1,000 nodes at academic institutions and industrial research labs distributed in more than 700 locations. Current node distribution as of November 2018 is shown in figure 1.1.

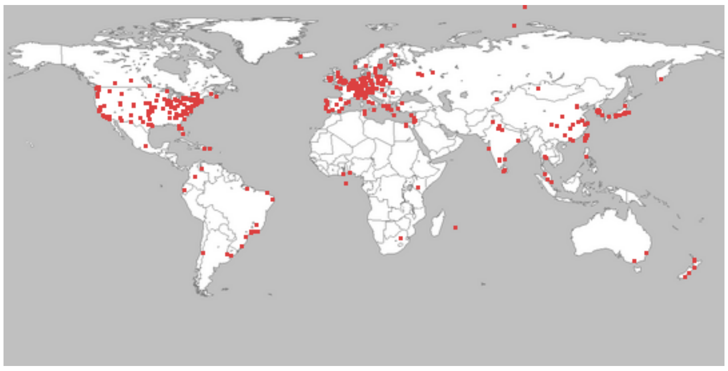


Figure 1.1: Geographical distribution of Planet-Lab nodes as of November 2018 [36].

This platform can be used to run any research project that is accepted by a Principal Investigator (PI) that is responsible for a Planet-Lab node. Each node corresponds to a physical computer that is leased to the Planet-Lab project. The institution must provide the actual hardware, housing space and Internet connection. Then, once the Planet-Lab operating system image is deployed and the node is started, it is ready to be allocated by any Planet-Lab user. Users can be assigned a virtual machine—or LXC container in newer versions—per *slice* (project) per node. They are given full software access to their slice, in which experiments can be deployed and executed. Actual node availability depends on external factors (connection quality, support team response) that may impede the same exact nodes to be used in different experiments consistently. As most nodes are located in academic facilities, it might not be possible to represent non-academic Internet connections[2] accurately. On the other hand, one of its strong points is the rapid availability of geographically distributed resources.

Another project that exploited this paradigm is Ono [37], a plugin for the Vuze BitTorrent file transfer client that prefers connections to nodes that are closer (in terms of communication latency), rather than using the default random selection that is defined in the BitTorrent protocol. Ono collected user data for further analysis, which was used to study the impact of this modification to the BitTorrent protocol. It leveraged the automatic plugin update feature in Vuze to push modifications to its peers. This project was discontinued in 2008.

The FIT IoT-LAB [25] Testbed consists of many IoT devices in geographically distributed sites with an unified administrative domain. The nodes are distributed in different sites which are located throughout France. The platform gives “bare-metal” access to IoT nodes of different types and processing power. Many devices have sensors that can be remotely queried, and some of them even allow remote login through SSH. The latter have enough memory to run full operating system images in which additional software packages can be installed.

1.1.2 Simulation

In this paradigm, a prototype or a model of the application is executed on top of a model of the environment. This approach enables the researcher to analyze questions about the system without having access to the actual environment or the actual application. The reliability of the results depend on the validity of the underlying models. In particular, inaccurate platform descriptions, missing hardware models or incorrect assumptions in network models could lead to wrong conclusions in using this approach.

A Discrete-Event Simulation (DES) is one in which the state variable changes only at a discrete set of points in time [38]. This is an approach that is a good match for computer network simulations, as the state changes when packets are generated. Most well-known simulators for distributed systems are discrete-event simulators.

Ns-3 [39] is a widely-used full-stack detailed discrete-event simulator designed for network applications. It is related to ns-2, another popular simulator that was active in the last decade. Ns-3 superseded it, but it is not backwards-compatible with ns-2. The older version exposed an Application Programming Interface (API) in the Tcl programming language, whereas ns-3 uses C++ for simulations. In order to use ns-3, a simulation has to be programmed using the simulator API and compiled to generate an executable. Building a simulation usually involves the following steps:

1. Create node entities and configure which network stack will be used.
2. Create switches and links as appropriate to match the input topology.
3. Set up link latency, bandwidth and other options, depending on the physical layer that is used.
4. Set up IP addresses and any routing configuration.
5. Launch application simulations in the desired nodes.

In the case of some typical topology families, helper classes are used to reduce the configuration overhead that is required to build a complete simulation. However, ns-3 requires a great amount of detail—down to the physical layer—to describe the system. This type of simulators is very difficult to parallelize, as it requires a centralized clock. Thus, the complete distributed system is executed in a serialized way, which could result in unmanageable execution times for big simulations. Applications that run on the nodes have to be modeled for the simulator, as ns-3 does not provide mechanisms to transform or adapt applications directly. An alternative to building the application code from scratch is an extension called Direct Code

[2] For instance, residential Internet connections might behave differently due to upload and download asymmetry, bandwidth restrictions and other usage constraints.

Execution (DCE) [40], which wraps C library calls to be simulated by ns-3. However, not all system calls are implemented in this virtual C library and is therefore not suitable for all applications.

SimGrid [41] is another widely used discrete-event simulator. It is aimed at simulating large-scale distributed systems efficiently. In order to scale simulations on a single node, allowing up to thousands of simulated nodes, SimGrid implements a simplified communication model with precomputed paths. The traffic flow is estimated using a configurable network model that can be tailored to a particular use case. Simulations are defined in a program that is compiled and linked against the SimGrid libraries, similarly to ns-3. SimGrid provides bindings that allow simulations to be written in several languages such as C, C++ and Java. Network topologies and process-to-node allocation can be defined directly from the simulation code or using XML configuration files:

- *Platform file*: Defines the network topology, i.e. node definitions, links and switches.
- *Deployment file*: Defines which processes are executed in which of the nodes that are defined in the platform file and when those processes should start running.

Using XML files gives the user more flexibility as it is not required to modify the source code of the simulation to change the network setup or other simulation parameters. The SimGrid API for communication uses an abstraction called *mailbox*. A mailbox is defined with a name, and it can be used as a destination for a message. This is used instead of IP addresses and TCP sockets, and therefore most existing network applications have to be adapted to be able to build SimGrid simulations. A notable exception is MPI applications, as SimGrid it provides an online mechanism, called SMPI [42], to execute them directly by wrapping MPI calls to the simulator engine. This API allows MPI applications to be ported to SimGrid simulations easily if the source code is available.

Finally, a simulation tool that is not a discrete-event simulator is Dimemas [43], a performance analysis tool for MPI applications, which is able to simulate offline execution on a given target architecture. Targets range from single- or multi-core networked clusters to heterogeneous systems. Performance is calculated by replaying the execution trace of an application on a built-in simulator. Similarly to SimGrid, this approach relies on a particular implementation of MPI primitives and the communication model itself.

1.1.3 Emulation

The actual application is executed in a simulated environment. A network emulator is often called a *WAN simulator*, as it defines an environment in which a networked application can be executed locally or in a local network, simulating features of a global network. The environment can be controlled through classical virtualization techniques, or a controlled artificial load can be injected onto real resources such as network links and CPUs according to a given experimental scenario.

Let's suppose we want to define a network scenario that consists of two virtual nodes, *A* and *B*, that are far away from each other. The latency from *A* to *B* is λ. Both nodes execute a piece of software that results in data being sent from *A* to *B*. In order to define this scenario in a network emulator, we first require physical resources to be allocated for each node. Depending on the emulation tool, a virtual node could be allocated in a full node, virtual machine, container or normal process. Additionally, the network view of each virtual node has to be configured to match the virtual topology. This includes the routing tables and the characteristics of the virtual links. In this example, node *A* should be able to reach node *B* in a single hop. If the network topology was more complex, requiring additional hops to reach the destination, this would also have to be taken into account by the network emulator.

Let the latency of the physical network be ϵ, a link of latency λ is emulated by sending it to an intermediate node *C* that will delay forwarding as much as λ. This is shown in figure 1.2. This approach requires that $\epsilon \ll \lambda$ in order not to introduce excessive delays that could result in the emulated latency being too different to the expected value. Using the same idea, it is possible to emulate packet loss, reordering, duplication and bandwidth limitation. Several network emulators replace the intermediate node by a packet queueing discipline that introduces the requested delay before leaving the sender. This requires operating system support, such as the NetEm [44] network emulation facility in the Linux kernel.

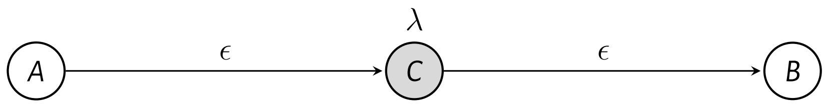


Figure 1.2: Emulation of a link with latency λ in a virtual topology of two nodes in which the latency of the physical nodes is ϵ.

The main advantage of this method is that it is possible to analyze changes in the application or the software stack and their impact on the overall performance of the system. However, testing in a large-scale environment might require a huge amount of computational resources that could render this approach unfeasible or limited to a small scale. A possible approach to improve the scalability of these systems is the use of *time dilation*. This technique fakes the clock source of the virtual nodes to make it seem that time advances at a slower pace than in real life. This approach has the same advantages as other network emulation tools, while allowing to model links at a bigger scale with better fidelity, at the expense of running time. A disadvantage of this technique is that experiments could possibly take much longer than in real life. One network emulator that implements time dilation is SELENA [45].

Another example is ModelNet [46], a network emulator that enables the execution of unmodified distributed applications on computer clusters. As the tool dates from 2002, it has been conceived and designed to use the operating system infrastructure that existed at that time. In order to emulate arbitrary networks, it handles network traffic at the packet level, but the underlying operating system APIs have changed with time, rendering it unusable on current software [45]. ModelNet defines two types of nodes: *edge nodes* and *cluster nodes* (or *core nodes*). Edge nodes run the actual instances of the target application, while cluster nodes provide the mechanism to route traffic according to the emulated topology. Traffic routing is achieved by piping the network flow to a kernel module which runs on each cluster node, modifying packet headers as required. Processes are executed directly on the edge nodes, replacing the socket operations with a custom library that makes sure that traffic is correctly redirected to the cluster nodes. An edge node may execute several processes, whose resources (e.g. CPU time, memory) are not isolated from each other. This solution depends on a kernel module that is restricted to the FreeBSD platform[3] and is difficult to maintain. Additionally, ModelNet has a dependency on custom versions of `gexec` and `authd`, two legacy components of the Ganglia Monitoring System that were used to launch applications remotely using public key authentication. This has been superseded by the SSH protocol in most modern tools. No development activity has been found after 2008.

Mininet [47] is a tool that emulates Software-Defined Networks (SDNs) in a single host. It leverages network namespaces and traffic shaping technologies of the Linux operating system to instantiate arbitrary network topologies. This tool requires all virtual nodes to be instantiated on a single host. Experiments are written in Python scripts that use the Mininet API to create topologies and execute commands in a reproducible way. Wette *et al.* [48] extend Mininet to span an emulated network over several hosts, using the GRE protocol to tunnel traffic from an emulated local network to another. Network partitioning is done by precalculating which namespaces should be assigned to each physical host. A popular graph partitioning tool called METIS is used for this task [49].

Emulab [50] is a shared testbed that can be used to instantiate arbitrary network topologies on a cluster partition. Free access to run network research projects on the testbed at University of Utah can be requested by members of the academic community. Experiments are defined using ns-2 scripts with additional

[3]An initial port for the Linux kernel existed in alpha version. However, it is already outdated as the kernel ABI has changed significantly since 2002.

commands that are specific to Emulab. These scripts, written in Tcl, allow network topologies to be defined and commands to be executed on arbitrary nodes. One restriction with relation to the network topology is that it cannot have any loops. Routing tables are defined statically, even though, by design, it is possible to support additional routing strategies. Link characteristics are simulated using additional intermediate nodes that run the FreeBSD operating system. These nodes are configured as transparent bridges that shape traffic according to the experiment settings. Intermediate node allocation and setup is automatically done by the experiment allocator script, and is completely transparent to the user. The allocator script also defines a Virtual Local Area Network (VLAN) to isolate traffic for that experiment from the rest of the cluster. This requires switch support. It is possible to self-deploy a local Emulab instance to have better control of job allocation and bandwidth usage, although it requires very specific programmable switching hardware[4] that might be too expensive or difficult to acquire.

1.2 The Fog/Edge Computing Model: Moving Computation down the Cloud

In the last years, the Cloud Computing model has emerged as an alternative to owning and managing computing infrastructure. Cloud Computing relieves the need of managing and supporting large and complex data centers. It also provides resources as needed, giving the flexibility to pay only for what is used. This technology provides an efficient solution to serve several types of applications, such as web servers, databases, storage and batch processing. It is also more inexpensive as a whole due to the economies of scale of big data centers.

Offloading is the technique used to partially or completely delegate a computation to a remote site. This allows the application to increase resource utilization as the complexity of the task grows.

As the cloud is usually farther away from the clients, latency-sensitive applications could suffer from performance degradation in this setup. The Fog Computing model is defined by Bonomi *et al.* [5] as a highly virtualized platform that provides compute, storage and networking services between end devices and the cloud. This model allows certain services to be offloaded to end nodes, thus reducing latency requirements, while improving in terms of system elasticity, defined as the ability to adapt to workload changes [6]. Going further, the proliferation of IoT devices with increasing computing power has given an identity to nodes at the edge of the network, defining the Edge Computing model [7]. These deployments require mobility support, geo-distribution, and low latency. In this model, edge nodes cooperate among themselves to provide results at lower response time or bandwidth requirements, while improving data safety and privacy with respect to handling processing at the cloud. As the number of nodes at the edge increases, it is also becoming increasingly more difficult to serve all clients at the cloud with this infrastructure in terms of server load and required bandwidth. End nodes are usually slower and have worse connectivity than the main infrastructure, but both aspects have been steadily improving, enabling execution of more and more applications.

1.2.1 IoT

IoT was first coined by Kevin Ashton in 1999 in the context of supply chain management [51]. The term has since then been extended to other applications, such as healthcare, utilities and transport [52]. The RFID group defines it as *the worldwide network of interconnected objects uniquely addressable based on standard communication protocols*. In this context, *thing* also has many definitions: it mainly refers to the active participants in business, information and social processes in which they can interact and communicate

[4]VLANs are configured by sending Simple Network Management Protocol (SNMP) commands to the switch. This is not standard and thus requires support for specific switch models. Only a handful (notably some Cisco models) are supported.

among themselves and with the environment, exchanging data while reacting autonomously to external events.

The paradigm of IoT defines a vision of computing in which most objects that surround us will be connected to a network of some sort [22]. This results in the generation of vasts amounts of data that have to be stored, processed and presented in an efficient and user-friendly way. This model is entangled with other paradigms, such as Cloud Computing and derivative families like Fog and Edge Computing. IoT builds upon the proliferation of different types of sensors and already deployed wireless networks, with a vision that goes beyond traditional computing to the realm of ubiquitous computing: everyday objects and appliances that are connected to each other and to other infrastructures in a seamless way, *disappearing* from the consciousness of the user. Gubbi *et al.* propose the following demands to achieve this goal:

1. A shared understanding of the situation of its users and appliances.
2. Software architectures and pervasive communication networks to process and convey the contextual information.
3. The analytics tools in IoT that aim for autonomous and smart behavior.

A critical issue for the success of IoT is to be able to uniquely address each object in a reliable, persistent and scalable way. The current widely deployed addressing scheme that is used on the Internet is IP version 4 (IPv4). However, this solution has limitations, as there is a severe address restriction to uniquely identify every sensor or device. This is partially solved by IP version 6 (IPv6), but the penetration of this technology is still not up to par with that of IPv4, imposing hurdles to the universality of this solution. Moreover, some types of sensors do not have the capabilities to run a full IP stack due to resource constraints, leading to the development of lightweight versions that fit into these devices.

As the capabilities of IoT devices increase, new types of applications emerge that could benefit from this paradigm. An existing effort to use distributed resources for computing is volunteer computing [23]. In this model, participants give up their idle CPU power to compute smaller chunks of a problem that cannot be easily solved individually. This has been proven to work for certain embarrassingly parallel problems [24].

1.2.2 Cloud Computing

Armbrust *et al.* [53] define Cloud Computing as *both the applications delivered as services over the Internet and the hardware and systems software in data centers that provide those services*. The data center hardware and software is what is referred to as a *cloud*, which can be used in a pay-as-you-go basis (public cloud) or sold as a platform (utility computing). Moreover, if the cloud is internal to a business or organization, it is called a *private cloud*.

The service models for Cloud Computing are categorized as a lower level Infrastructure as a Service (IaaS), in which the users have access to the virtualized infrastructure, and higher level Platform as a Service (PaaS) and Software as a Service (SaaS) [54], which are defined as follows:

- IaaS: Providers of IaaS offer infrastructure resources, such as physical nodes or (more often) virtual machines. This model hides the actual location of the resources to the user, while giving the user the ability to manage the complete operating system and middleware services that run on top of that infrastructure. Server housing, storage and the networking infrastructure are also managed by the service provider. The user is responsible for keeping their services running and provide a security upgrade cycle for the service itself and the underlying operating system. The user cost of IaaS depends on the amount of resources allocated and consumed. Some examples of this model are Amazon EC2 and S3 and OpenStack services.

- PaaS: This is a development environment for application developers. Cloud providers offer a full computing platform, from the low level infrastructure to the actual operating system, middleware and runtime libraries. This allows the users to deploy their applications without managing the cost and complexity of the underlying software and hardware layers. Some examples of PaaS are Microsoft Azure, Google App Engine, Heroku and OpenShift. A typical use case is to deploy and manage scalability of web applications.

- SaaS: In this model, users gain access to application software and databases. Cloud providers are responsible for managing the complete stack, including the service itself. This greatly simplifies maintenance and support for the end user. Moreover, as the infrastructure is directly managed by the cloud provider, it is easy to scale a service to serve more users during peak times. The pricing model for SaaS is usually a monthly or yearly flat fee per user. Some examples are: Google for Work (provides standalone instances of the Google services for an organization) and iCloud (used as a remote storage solution for users of Apple products).

Several novel aspects are defined in the cloud computing paradigm in terms of hardware provisioning and pricing:

- A great quantity of resources are available on demand, eliminating the need for users to plan far ahead for provisioning.
- As cloud users don't manage their infrastructure themselves, it allows them to start small and expand as needed, without requiring a big initial investment.
- Pay-as-you-go results in reduced costs if resource needs are sporadic.

From the point of view of the provider, economies of scale result in greatly reduced costs when compared to the users having to implement their infrastructure in a smaller scale themselves. Multiplexing user load at idle times—while also being paid for it—allows the existing infrastructure to be used more efficiently in terms of capitalizing those investments.

1.2.3 Fog Computing

Resources in both the Cloud and Fog Computing models can be categorized in three types: Compute, storage and networking. Compute is any node that does computation, storage represents any type of persistent storage and networking refers to the underlying network. Fog extends these categories by including resources of the previous types that are located on the edge of the network [5]. The location of a resource is hidden from the user through the use of virtualization techniques, but resource scheduling plays nonetheless an important role in providing an acceptable quality of service in this model. The nodes at the edge are compute and network resources that are located between data sources and cloud data centers. They provide location awareness and a lower latency, which enables other type of applications such as gaming, video streaming and Augmented Reality (AR). Other characteristics include:

- Geographical distribution: Unlike the cloud, services and applications for the Fog have to be widely distributed. For instance, it could satisfy high quality streaming to moving vehicles through devices that are located along the road.
- Large-scale sensor networks to monitor the environment: The Smart Grid is an example of a distributed system that requires distributed computing and storage solutions.
- Very large number of nodes, as a result of the deployment distribution.
- Support for mobility: Most use cases involve communication with mobile devices, and therefore it is required to decouple host identity from location. A possible solution is to use a distributed directory system.

- Real-time interactions, instead of batch processing.
- Predominance of wireless access.
- Heterogeneity in device types and environments.
- Interoperability and federation: Different providers have to be able to cooperate with each other to provide certain services, such as streaming. This requires a mechanism to interoperate, and federated access across domains.
- Interplay with the cloud to receive and process data close to the source and reduce the cloud's workload.

This paradigm has been originally designed with a subscriber model in mind (e.g. Connected Vehicles, Smart Grid, Smart Cities) and is thought to boost new forms of competition and cooperation between service providers. However, it is still not clear what implications it will have in the way compute and other types of services are designed and provided.

1.2.4 Edge Computing

The Edge Computing model is complementary to Fog Computing and refers to the ability to perform computation at the edge of the network on downstream data on behalf of the cloud and upstream data on behalf of IoT services [7]. Both models are interchangeable, but have a different focus: while Fog Computing is more interested in the infrastructure side, Edge Computing is more focused on the IoT side.

This change in paradigm has implications such that IoT devices are not only consumers of data, but also producers. The main design factor is that computation should happen as close to the source as possible.

This model has benefits that include reduced energy consumption, an improvement in response times and reduced workload in the main cloud infrastructure.

1.3 MPI: The De-Facto Standard for Distributed Scientific Computing

The standard API in scientific parallel applications is MPI [9, 10, 11]. It has been designed to provide an abstraction to handle data exchange and synchronization among execution units irrespective of whether they are located on the same node or on different ones. Message passing is a very powerful API for developing parallel programs, which also works at a very low level. This means that the user has to take care of many details explicitly. Parallelizing an application—transforming a serial program into a parallel one using MPI—involves several steps that include changing and/or replicating data structures, explicitly distributing data among processes, collecting data or synchronizing, giving MPI the nickname of "the assembly language of parallel programming". Those steps aren't incremental, requiring a full rewrite of the original sequential code in many cases.

The basic execution unit in MPI is the *process*, which refers to a particular instance of a parallel program. Processes in MPI can be executed in the same node or throughout the network transparently. The library takes care of defining which is the most efficient way to communicate depending on the destination of the message, hiding these details from the programmer. For example, if two processes are located in the same computer, it is possible to use a much faster shared memory buffer to communicate, but if they are in different nodes, messages must be sent through the network. Network messages can also be handled differently depending on the transport technology (e.g. InfiniBand, TCP, SCTP, Omni-Path). The availability of network technologies depends on the implementation, supported hardware and application arguments, but it is transparent to the user. In order to send a message to a particular process, the programmer need only know an identifier that represents the destination, as MPI takes care of efficiently delivering the message.

The basic operations defined on MPI can be categorized in one-to-one, one-to-many, many-to-one and many-to-many communication. One-to-one or **P2P** operations simply include sending and receiving data from a particular process. This can be resolved asynchronously or synchronously depending on the application needs. Asynchronous sends allow the programmer to reduce idle times derived from communication delays, as the process resumes execution as soon as the message is scheduled for dispatch by the operating system. Any additional computation that is handled during these times may result in an increased parallelism, given that it is possible to do useful calculations while the communication is in progress. The receiver end can also benefit from asynchronous receives, given that it can do useful operations while it is expecting data from another process. The programmer is responsible for using this feature efficiently. In the case of synchronous operations, the implementation may block the process while it is in progress, but this responsibility is not delegated to the operating system[5]. Moreover, two types of synchronous sends are defined in the standard [55]:

- Regular synchronous send (`MPI_SEND`): the implementation will return as soon as the memory buffer is available for reuse (i.e. the data is already in kernel space). This could happen before the data has arrived at destination.

- Blocking synchronous send (`MPI_SSEND`): it will always wait until the message has arrived.

The receiving end is always blocking in the synchronous case, as it requires its memory buffer to be filled with actual data. Additionally, most MPI implementations define two protocols for P2P sends to be used depending on the amount of data that is to be transferred:

- *Eager*: for short messages, minimizes interaction between sender and receiver as it includes MPI envelope information eagerly with the data. This information is used by MPI to identify the message. The eager protocol improves the latency for short messages.

- *Rendezvous*: for long messages, sends a *ready* message before actually starting the send. This allows the receiver to allocate the required resources before the data is transferred, resulting in better bandwidth for long messages.

These protocols are shown schematically in figure 1.3.

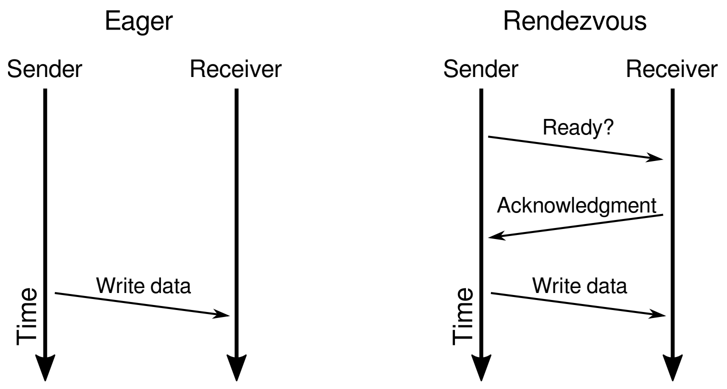


Figure 1.3: MPI messaging protocols.

One-to-many, many-to-one and many-to-many or **collective** operations are useful for propagating data to worker processes and gathering partial results. Additional commonly used collective operations include

[5]In the case of TCP sockets, non-blocking sockets are used even for synchronous communication.

mechanisms for synchronization barriers and doing one-to-many and many-to-one operations in parallel in all processes. Collective operations are implemented as a series of point-to-point messages. Several implementations are available for each collective type in order to reduce message duplication and total propagation time. Which strategy is better depends on the transport layer, the size of the messages and the number of destinations.

These characteristics make static analysis of the traffic profile of MPI applications a very complex task. It depends on the actual implementation of collective messages, the message protocol (which may in turn depend on the size of the message), the use of blocking or non-blocking communication and the overlapping of computation and communication, that is closely related to the hardware in which the application is executed and the network technology.

Most implementations are targeted to be used on HPC clusters, assuming full node connectivity and implementing optimizations for collective operations that work more efficiently in homogeneous environments. They are also very sensitive to network changes, bringing the computation to a halt if just one node disconnects or changes its network location[6].

The initial open implementation of the MPI-1 standard was MPICH, from Argonne National Laboratory, which was initially released in 2001. Also LAM/MPI and, later, Open MPI offered open alternatives. MPICH is the basis for many implementations of MPI, and it is still being maintained as of 2018, currently covering the MPI-3.2 standard. MVAPICH is another implementation that is directly based on MPICH, with an emphasis on InfiniBand support.

Commercial implementations also exist, including those from HP, IBM, Intel, Cray, Microsoft, and other companies that have participated in the HPC world.

1.3.1 Architectural Overview of MPI Implementations

Writing efficient MPI code is fundamental to take the most out of an HPC infrastructure. Special care must be taken to reduce idle times and unnecessary data propagation. However, it is not just the user application that needs to be properly optimized. The MPI library itself has to be carefully conceived to make efficient use of the operating system facilities and the available hardware.

In the next paragraphs, we discuss design decisions of two of the main open implementations: Open MPI and MPICH.

Open MPI

This implementation has a layered design [56]. Its architectural view is shown in figure 1.4.

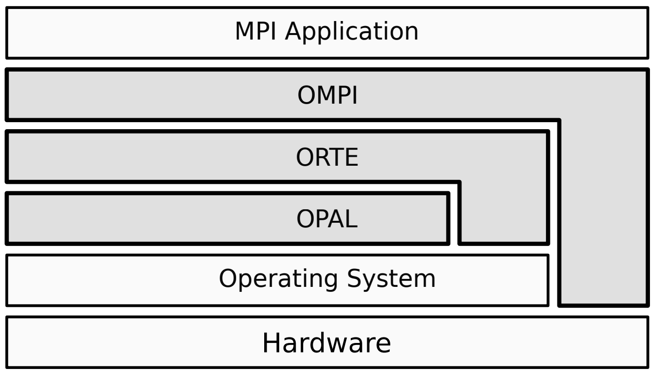


Figure 1.4: Architectural view of Open MPI showing its main layers.

At the top lies the **MPI application** layer. This is the user application that makes use of the MPI library.

[6]Except FT-MPI [12].

Below, the library provides **Open MPI API (OMPI)**, which encapsulates the logic semantics of the implementation of the API calls.

Open Run-Time Environment (ORTE) is, as the name implies, the runtime environment, which manages process creation and destruction throughout different nodes and operating systems. ORTE has support for different backends to launch processes in other nodes. The default backends are Remote Shell (RSH) or SSH, but other specialized options exist for cluster environments, such as Torque or SLURM, which allow process accounting and scheduling in multi-user systems.

Open Portable Access Layer (OPAL), the bottom layer, includes implementations of basic data structures and handles network address discovery, use of shared memory buffers, and other low-level configuration that is abstracted away for portability. This layer talks to the operating system directly. Synchronous or blocking API calls are defined internally as non-blocking, while this layer blocks explicitly if required and provides efficient handling of message data.

In some cases, upper layers have direct access to the lower ones for optimization purposes.

Many modules in all layers can be implemented in different ways depending on the target platform. Open MPI defines a plugin architecture which is called Module Component Architecture (MCA), that provides a mechanism to efficiently handle composition of different module implementations, giving the user maximum flexibility at each layer with a very low overhead. This allows Open MPI to support different types of process launchers, network transport layers (with their respective buffer handling mechanisms), process creation APIs, timers, implementations of collective messages, among others. In discussing MCA, the following terms are used:

Framework A public interface that is designed with a particular purpose in mind, but can be implemented in multiple ways in the code base (e.g. `btl`, the Byte Transport Layer, which is used to send and receive P2P messages on some types of networks).

Component (or *plugin*) An implementation of a Framework's interface (e.g. `tcp` and `openib` implement the `btl` framework for TCP and InfiniBand respectively).

Module An instance of a Component (i.e. Components' stateful information, such as keeping track of different TCP sockets in the `tcp` component).

Let's take the `coll` framework as an example, which defines the semantics of each collective message. It is defined in the **OMPI** layer and it is used to determine how the information should be propagated in the network, depending on the collective message and the underlying network. There are several components such as `basic` (straightforward, non-optimized implementations), `cuda` (for CUDA support), `sm` (for processes in which a shared memory buffer is available) and `tuned` (optimized for a particular network configuration). By default, if one process is connected to another using the `tcp` component, the `tuned` component is used to provide the `coll` framework. This `tuned` component defines several rules for each collective message that can be overridden at startup time using configuration switches [57]. In the case of the broadcast algorithm, the MCA variable `coll_tuned_bcast_algorithm` indicates which algorithm should be used with a number ranging from 1 to 6:

1. **Binary tree**: Defines a binary tree and forwards data following that structure.
2. **Split binary tree**: Defines a split binary tree and forwards data following that structure.
3. **Pipeline**: Defines a chain (see below) with a fanout of 1 (i.e. sends data following a linear chain).
4. **Chain(*N*)**: Sends data following a chain with a configurable fanout of ***N***.
5. **Basic linear**: Broadcasts data one node at a time, in linear order, using non-blocking sends.
6. **Binomial tree**: Defines a binomial tree and forwards data following that structure.

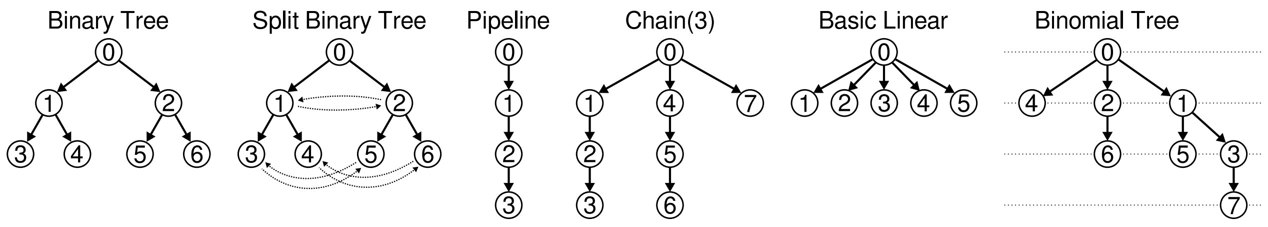


Figure 1.5: Traversal structures for each broadcast algorithm in MPI.

These structures are preconfigured depending on the communicator. Each process calculates its place in the structure (i.e. parent and/or children nodes) independently. Only its own rank number and the configuration setting are required. The resulting digraphs for each algorithm are shown in figure 1.5. The orientation of the links corresponds to the order in which the messages are propagated, starting from the root process (rank 0 in the examples).

The definition of an additional algorithm involves the following steps:

1. Increment the `coll_tuned_bcast_algorithm_count` constant.

2. Modify the `ompi_coll_tuned_bcast_intra_do_this` to call the new algorithm.

3. Define a build function that is executed when the module is initialized to create the tree structure (`ompi_coll_tree_t`).

These optimizations are not limited to just the broadcast algorithm. The rest of the collective operations also allow multiple implementations to be tuned depending on the use case. Being able to use more intelligent data propagation algorithms is important for Fog/Edge environments, as information of the underlying topology could be used to do these operations more efficiently.

This is just one example how components are used in this library. Open MPI provides more than 100 components through the MCA.

MPICH

This library shows a number of similarities to Open MPI in its design decisions. It also uses a layered design. It is thought as a "framework" for MPI implementations, in which many components can be replaced with alternative implementations that adapt to particular use cases. Recall MVAPICH, which is an implementation based on MPICH that comes in a number of flavors that support different transport technologies, mainly InfiniBand, but also Intel Omni-Path, SR-IOV and others.

Similarly to Open MPI, MPICH offers bindings to several programming languages in its top layer, the **MPI Interface**. Below, the MPI functions are implemented in terms of **Abstract Device Interface (ADI)** functions and macros. This interface is defined for portability reasons and provides functions to send or receive data, buffer messages and query environment information. There are several modules that implement the ADI. The **Channel Interface (CH)** enables the use of alternative implementations for the transport layer. It supports different network hardware and platforms. TCP/IP support exists as module of **Chamaleon**, which implements CH. Finally, the **Process Manager (PM)**, called **Hydra**, is used to manage processes. This module works similarly to ORTE in Open MPI. It also supports multiple process launchers such as SSH and RSH.

The main difference in flexibility with respect to Open MPI is the lack of a runtime plugin system. Modules are compiled together depending on the target hardware to create different flavors of the library.

1.3.2 Structure of an MPI Program

MPI provides an API with bindings for most widely spread programming languages, including but not limited to Fortran, C, C++, Python and JavaScript. In the case of the C version, a header file called `mpi.h` includes all function and data structure definitions that are exposed to the user.

All MPI identifiers are prefixed with `MPI_`, making it easy to differentiate them from user symbols.

In MPI applications, each process has an identifier which is called *rank*. It is used as the location of the process (the source or destination of a message), hiding its real location from the programmer. For collective messages, ranks can be grouped together using an abstraction called *communicator*. By default, a main communicator is defined by the process manager. It is called `MPI_COMM_WORLD` (or the world communicator, using language-agnostic notation) and includes all ranks. The world communicator is always available and can be used to create new groups that include only a subset of nodes. Note that the ranks of a process in two different communicators need not agree.

The main functions the API provides are the following:

MPI_Init Initializes the MPI system, resolving the initial communication and client setup. No other MPI instruction can be executed before it.

MPI_Finalize The last instruction that should be called, also a synchronization point for all processes.

MPI_Comm_size Returns the size of a communicator.

MPI_Comm_rank Returns the rank of the calling process in a particular communicator.

MPI_Send Sends buffer to a rank, blocking the calling process while it is in progress.

MPI_Recv Receives buffer from a rank, blocking the calling process while it is in progress.

MPI_Isend Sends buffer to a process asynchronously.

MPI_Irecv Receives buffer from a process asynchronously.

MPI_Wait Waits on an asynchronous send or receive.

MPI_Barrier Blocks all processes in a communicator until they have reached this point.

MPI_Bcast Broadcasts buffer to all processes in a communicator. The source of this message, the *root* process, is explicitly specified by rank in this function call.

MPI_Reduce Gathers data from all processes in a communicator, reducing all values to a single one by using an aggregation function. The result is stored in the *root* process.

MPI_Scatter Distributes data from the *root* process among all other processes in a communicator. The buffer is split in n chunks of the same size, one for each member of the communicator.

MPI_Gather Gathers data from all processes, opposite operation to **MPI_Scatter**. The result is stored in the *root* process.

MPI_Allreduce Efficiently performs a reduce on every process. This is functionally equivalent to doing an **MPI_Reduce** and then an **MPI_Bcast**.

MPI_Allgather Results in all nodes gathering data from everyone else in the communicator.

MPI_Alltoall Sends data from all processes to all processes.

A minimal example is shown in figure 1.6: a C implementation of an "MPI hello world" program. This code initializes MPI (line 8), requests its rank in the world communicator (line 9), prints it out (line 11) and finalizes (line 13). The compiled program is to be called using the process runner (usually `mpirun` or `mpiexec`), which will communicate to the remote nodes (if any) and create the requested number of instances.

```c
 1  #include <stdio.h>
 2  #include <mpi.h>
 3
 4  int main(int argc, char **argv)
 5  {
 6     int node;
 7
 8     MPI_Init(&argc, &argv);
 9     MPI_Comm_rank(MPI_COMM_WORLD, &node);
10
11     printf("Hello World from node %d\n", node);
12
13     MPI_Finalize();
14
15     return 0;
16  }
```

Figure 1.6: Minimal "MPI hello world" sample program in C.

From a point of view of the runtime, a typical execution scenario of a compiled MPI program in different nodes involves the following steps:

1. `mpirun` connects to every node and runs `orted` (ORTE daemon) everywhere. The execution fails if `orted` cannot be executed (e.g. if remote access is denied).
2. `orted` establishes connections between nodes. Not every node pair is connected at this point, some connections are established on-demand when the application is running.
3. An instance of the same program (process) is executed on each node by `orted` with appropriate environment information. The runtime defines environment variables to properly identify each instance. For example, the `OMPI_COMM_WORLD_RANK` variable shows the process rank in the world communicator for that instance. An MPI program doesn't access these variables directly. Instead, this information is processed by the MPI runtime and accessed using MPI calls such as the one at line 9 in the sample program.

Note that the same program is executed everywhere. In some cases, a networked filesystem is used to share data between nodes, which also includes the binaries that are to be executed. By default, the process runner searches for the binary using the exact location in which the local executable resides.

As the same program is being executed everywhere, their own rank is typically used to alter the control flow of the program for a particular instance. In some collective operations, the process rank is also important as it determines which process is the *root*. For example, in an `MPI_Bcast` call, all processes execute the same instruction with the same parameters, which will result in process ranked *root* sending data to the rest.

Benchmark code	Problem size	Memory (Mw)	Time (sec)	Rate (Mflop/s)
Embarrassingly parallel (EP)	2^{28}	1	151	147
Multigrid (MG)	256^3	57	54	154
Conjugate gradient (CG)	14000	10	22	70
3-D FFT PDE (FT)	$256^2 \times 128$	59	39	192
Integer sort (IS)	2^{23}	26	21	37.2
LU solver (LU)	64^3	30	344	189
Pentadiagonal solver (SP)	64^3	6	806	175
Block tridiagonal solver (BT)	64^3	24	923	192

Table 1.1: NAS Parallel Benchmarks Class A Statistics [21]

1.4 The NAS Parallel Benchmarks

The NAS program at NASA Ames Research Center was conceived, in the words of the organization, "*to provide the* [USA]*'s aerospace research and development community by the year 2000 a high-performance, operational computing system capable of simulating an entire aerospace vehicle system within a computing time of one to several hours*" [21].

In 1991, a set of benchmarks was developed as part of the NAS program to assess high-end parallel supercomputers [13]. This was important to help NASA select which new supercomputer was the best purchase, as technical information provided by the vendors and the scientific community often overestimated the capabilities of this type of hardware. At that time, only the Linpack high-end benchmark was available and, although useful, it was considered that it didn't fit the type of computing the applications at NASA's supercomputers did. One possible solution was to use actual large-scale code, but this posed difficulties to be evaluated in a broad range of parallel systems. Moreover, back then, there was no generally accepted parallel programming model, as this was years before MPI or other models were even designed. This situation led to initially design the benchmarks as "paper and pencil" benchmarks, i.e. a set of requirements in a technical document that were accurately described and standardized, including sufficient detail in the description of the algorithms and even which input data should be used. Since then, reference implementations came up that implement this specification using standard libraries and code extensions. With the advent of MPI, OpenMP and other parallel programming models, specific versions of the benchmarks were released for each of them. We will call the MPI version indistinctly NPB or NPB-MPI during the rest of this work. The original NAS Parallel Benchmarks (NPB) consisted of 8 problems, each focused on a particular scientific computing problem. They provide a good baseline to study different aspects of HPC applications, as they have already been thoroughly studied by the community [14, 15, 16, 17, 18] in different contexts. Case studies for NPB involve parallization efforts in other programming architectures and execution platforms. Five of them are "kernels" (EP, MG, CG, FT and IS) and the last three are simulated CFD applications (LU, SP and BT). The kernels are compact problems with an emphasis on a particular type of numerical computation. The CFD applications reproduce data movement and computation required in CFD code and other 3-D physical simulations.

Each benchmark can be instantiated on a particular problem size (or class) and node count (or size). Each class is predefined and the actual problem size depends on it and the benchmark itself. Table 1.1 shows the actual problem sizes and other relevant statistics for class A. The class names follow these criteria:

- **S**: small, for quick test purposes.
- **W**: workstation (equivalent to a single workstation in the 90's).
- **A** to **C**: defined as 4 times the size of the previous class.

- **D** to **F**[7]: defined as 16 times the size of the previous class.

NPB-MPI implementations have been programmed in Fortran, with the exception of IS being in C. An overview of each benchmark follows, including a brief analysis of the characteristics of the messages that each of them use. The data has been compiled from Bailey *et al.* [21] and our own analysis of execution traces that were processed using the **mpiP** MPI profiler [58].

EP EP stands for "embarrassingly parallel". This kernel has no significant interprocess communication, i.e. it does computation independently on every process, requiring a root process just for coordination and result gathering. This type of computation is called Bag-of-Tasks (BoT), and has interesting properties as each part of the problem can be scheduled for execution in any order, or even replicated for better resiliency to node failures. The objective of this benchmark is to measure maximum floating point performance. The problem itself consists of generating pairs of Gaussian random deviates according to a specific scheme. It is of interest as it is typical of many Monte Carlo simulation applications.

MG Implements a simplified multigrid kernel. The problem consists of computing four iterations of a multigrid algorithm, which is used to obtain an approximate solution u to the discrete Poisson problem

$$\nabla^2 u = v$$

on a $256 \times 256 \times 256$ grid with periodic boundary conditions.

It requires highly structured long distance communication and tests both short and long distance data communication, employing unstructured matrix-vector multiplication. Message sizes in this benchmark are highly irregular, although bigger messages are more common than smaller ones.

CG An implementation of the conjugate gradient method. It uses the inverse power method to approximate the smallest eigenvalue of a large, sparse, symmetric positive definite matrix with a random pattern of non-zeros. It features both long- and short-distance communication and irregular memory access. Message sizes in this benchmark are big for P2P communication and small for broadcast, while most of the communication time is spent on P2P sends.

FT Solves a 3-D Partial Differential Equation (PDE) using forward and inverse Fast Fourier Transforms (FFTs). The PDE corresponds to

$$\frac{\partial u(x,t)}{\partial t} = \alpha \nabla^2 u(x,t)$$

This is essentially how many "spectral" codes look like. It tests long-distance communication performance. FFTs are used in certain CFD applications, such as eddy turbulence simulations.

IS Performs a large integer sort, which is used in certain "particle method" codes. It tests both integer computation speed and communication performance. This is the only code from the benchmark set that is written in the C language.

LU Performs a synthetic CFD calculation by solving regular-sparse lower and upper triangular systems.

SP Tackles another synthetic CFD calculation which consists of solving multiple, independent systems of non-diagonally dominant, scalar, pentadiagonal equations.

BT Similar to SP, but the equations are block tridiagonal (with a 5×5 block size) in this case.

[7]Class F is only available in the multizone version (NPB-MZ).

1.5 Conclusions

In this chapter, we have introduced the theoretical framework in which the work of this thesis is emplaced. Our main research driver is whether it is feasible to execute distributed computing applications in heterogeneous network and CPU environments, so as to make use of an increasing number of geographically distributed interconnected devices.

We have described three paradigms and their associated tools to evaluate distributed systems experimentally: experimentation, simulation and emulation. Each paradigm has its own strengths and weaknesses. The experimentation paradigm requires access to the real system to instrument it, run it and analyze metrics using real data. Simulation, on the other hand, requires application and environment models, but may be evaluated using little resources or without access to the real system. Emulation is a middle ground in which the application itself is evaluated as-is, but the network environment is emulated using traffic shaping or other mechanisms. We have chosen the emulation paradigm for this work, as we believe it is the most viable approach to study systems that are evolving or still under development.

Then, the Cloud, IoT and Fog and Edge Computing taxonomies are defined to properly contextualize the necessity to distribute computation.

We have introduced the MPI architecture, which is standard in the HPC world. We have described the main API calls and the architecture and some design decisions of two well-known implementations, MPICH and Open MPI. Additionally, a benchmark set called NAS Parallel Benchmarks (NPB) is presented, which consists of eight parallel programs or “program kernels” that mimic components of parallel applications that solve CFD problems. In particular, our interest lies in the MPI version of NPB. These benchmarks will be used in experiments on heterogeneous environments throughout most of this thesis.

2

Methodology

In this chapter, we describe the methodology we have proposed for this thesis. It is tightly associated to a novel tool, SherlockFog, which we also present and analyze in detail in the following sections. We explore different features, usage examples and language definitions that conform the platform. We also discuss limits and some use cases. The contents of this chapter are fundamental to the rest of the work, as all experimental scenarios and results are based on what is described here.

2.1 Proposal

We propose a methodology to support the analysis and porting of distributed computing applications to be executed following the paradigm of Edge Computing. Our proposal focuses on the impact of different traffic patterns in applications, with an emphasis on MPI, as it is the most widely used API for message-passing distributed scientific computing. However, our approach is valid for other distributed programming models and applications, as it is shown in chapter 5.

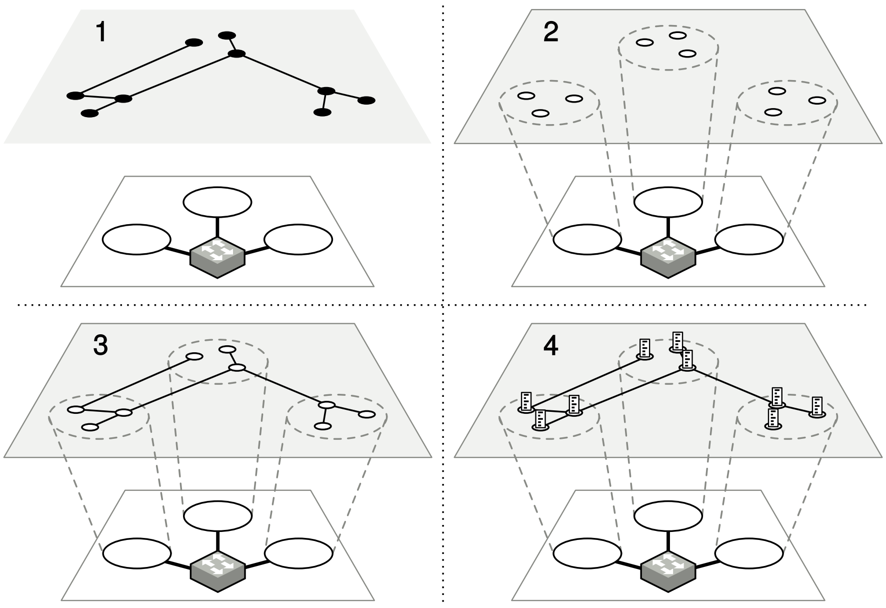


Figure 2.1: SherlockFog allows the user to analyze application behavior while varying network topologies and their properties in a reproducible procedure.

From a researcher perspective, given an application and one or more network scenarios and configurations, the user can define the settings programmatically and evaluate each of them using the tool. Figure 2.1 shows the process schematically. Each of the four diagrams in the figure corresponds to the following steps:

1. The user selects an application and a topology and creates an experiment script for SherlockFog to be deployed on a set of physical nodes. A host list is also to be provided to indicate SherlockFog where each virtual node should be instantiated.

2. SherlockFog is executed at the coordinator, which connects to every node and initializes network namespaces for each virtual node.

3. Virtual links are generated to match the input topology by creating virtual network adapters and assigning them addresses. Static routing is used to allow applications on each namespace to communicate with each other.

4. Application code is launched on the virtual nodes as required.

The tool allows the user to change network parameters during the run. This process can be repeated, comparing different topologies or input parameters. The output of the application is then analyzed, comparing its behavior on different scenarios.

2.2 SherlockFog

SherlockFog [59] is a tool that takes care of automating the deployment of a given emulated network topology and running experiments on top of it. In a nutshell, it *transforms* a script written in a custom language that defines a topology and the description of an experiment into hundreds or thousands of shell commands that achieve this goal. These commands are executed sequentially over one or several hosts. Its main focus is the execution and evaluation of MPI applications in non-standard configurations, with an emphasis on Fog/Edge Computing network scenarios, although other types of distributed systems can be emulated as well.

It makes extensive use of the `ip` tool—found on most GNU/Linux installations—to set up virtual Ethernet interfaces inside Linux Network Namespaces. The virtual interfaces are created by the `veth` command of `ip`, using the macvlan feature[1] in bridge mode. Macvlan allows a single real network interface to have different MAC addresses, each connected to a different “subinterface” that is managed independently. The kernel takes care of routing incoming traffic to the correct interface by looking up the destination MAC address of each fragment. A pair in the virtual network is connected by simply assigning IP addresses in the same P2P subnet[2] to both endpoints. Its traffic flows through the carrier of the host network interface. Each interface has not just a different MAC address but also a standalone configuration (e.g. its own name resolution dictionary, firewall, Address Resolution Protocol (ARP) and routing tables).

Routing is configured statically on every namespace in order to match the input topology. The static rules are generated using an algorithm that is user-configurable (see section 2.2.1). All virtual nodes act as routers, forwarding packets to its neighbors. Moreover, ARP is disabled on all virtual interfaces to prevent virtual nodes which are not neighbors in the virtual topology to find each other by sending ARP requests, which would result in the virtual topology being bypassed.

An SSH server is brought up on every container automatically to be able to launch MPI applications remotely. It runs on a different UTS namespace[3], whose hostname matches that of the virtual host. This feature is used to isolate the applications inside the emulated network as some MPI implementations check

[1]`https://hicu.be/bridge-vs-macvlan`

[2]A /30 network prefix.

[3]http://windsock.io/uts-namespace/

the hostname to define whether shared memory or a network transport should be used for communication. To the best of our knowledge, no other network experimentation tool takes this into account.

The virtual nodes are further isolated by using Linux control groups (cgroups), a feature that is used on other similar platforms, such as Mininet-HiFi [60], to improve the emulation fidelity. It is possible to assign CPU cores for exclusive access so that the client code that is executed in a virtual node is not migrated by the kernel at runtime.

It also allows MPI host files to be set up more easily using consistent names, the choice of real hosts notwithstanding. Name resolution is handled by generating appropriate `/etc/hosts` files for each namespace automatically. These files are bound by the `ip netns exec` command.

Finally, using the NetEm traffic control extension [44] via the `tc` tool, link parameters can be modified on a given virtual network interface's outbound port. The following sections define the features, implementation details and usage of the tool in more detail.

2.2.1 Main Features

The main features from a user perspective are the following:

- SherlockFog runs on commodity hardware, such as interconnected desktop computers in a university campus. No special interconnection technology or programmable switch is required, lowering the cost of ownership significantly with respect to similar solutions.
- A scripting language called *fog* allows to set up the topology, experiment parameters, and runs, enabling for reproducible experimentation.
- The tool can connect namespaces in the same physical computer or in different hosts, provided that the traffic that every host generates is reachable from the rest. This allows us to scale experiments by using hosts on different interconnected switches.
- It provides resource isolation (network, CPU and memory region) between virtual nodes that run in the same system.
- Application code can be executed unmodified. The user can execute open- or closed-source programs as they would in a real environment.
- It is possible to experiment with changes to the MPI library, such as broadcast implementations for Edge environments or features that make it more resilient to churn or changes in latency or bandwidth. As we are exploring MPI on non-standard network settings, our tool could be used as a testing framework for these use cases.
- It provides a mechanism to model mobility by changing bandwidth and packet loss of a link. This feature is also an important aspect in Fog or Edge Computing environments.
- The user can also set up a timer to change bandwidth, latency or execute a command after a fixed amount of time. This feature is implemented by launching a thread in the coordinator which executes the command on the target node when a timer expires. This is useful for analyzing temporary changes to topology settings in a reproducible way.

We show a sample script in the fog language in figure 2.2. In this example, four nodes named *n0* to *n3* (line 2) are initialized and connected sequentially, generating a "linked-list", while setting up a 5 ms delay between nodes *n1* and *n2* (lines 4–6). Line 8 configures IP addresses for all nodes and sets up static routing tables accordingly. Finally, lines 10–13 run the actual experiment: an MPI application is repeatedly executed as user *"myuser"* with argument *m* ranging from 10 to 100 in steps of 10, increasing latency between *n1* and *n2* on each step while saving its output for offline analysis. The `-f h.txt` command line option of `mpirun` defines the host file to `h.txt`: this file can be hardcoded using the names of the virtual nodes, as SherlockFog takes care of correctly setting up name resolution in the virtual infrastructure.

```
### node def
for n in 0..4 do def n{n}
### connect nodes
connect n0 n1
connect n1 n2 5ms
connect n2 n3
### build
build-network
### run exp
for m in 10..101..10 do
  runas n0 myuser mpirun -f h.txt ./p {m} > {m}.log
  set-delay n1 n2 {m}ms
end for
```

Figure 2.2: Example SherlockFog experiment script to launch a virtual topology of four nodes and execute an MPI application in different network conditions.

Topology Generation

Setting up a big network by hand using SherlockFog's custom language might become a tedious task. Thus, it is necessary to provide the tool with a mechanism to use generated topologies.

We have implemented a topology generator that can be used to convert between graph formats, including SherlockFog's language, and to generate some well-known topologies. The generator supports the following formats:

- **fog**: SherlockFog's own format can be read and parsed to be converted to other formats for visualization, or used as the output format to create scripts for the tool.
- **GML**: The Graph Modelling Language can be used as an exchange format. It is supported by Python's NetworkX library for graph processing and several other graph visualization tools. It can be used as an input or output format.
- **PDF**: Only as output format, uses NetworkX to draw the graph.

Additionally, it can be used to generate these types of graphs:

- **Barabási-Albert**: Random graph generated using the Barabási-Albert model for scale-free networks with preferential attachment with parameter m.
- **Isles**: Two clusters of nodes (star topology) connected through a single path. See experiments in section 3.2.2.
- **Fully Connected**: A complete graph (K_n).
- **Linked list**: Node i is connected to node $i + 1$ (except for the last one), forming a linked list.

- **Ring**: Node i is connected to node $i+1 \mod n$, forming a ring.
- **Star**: A single node 0 is connected to every other node i, forming a star.

Command Line Interface

SherlockFog offers a command line interface to remotely instantiate experiments. It requires two main input files: a host list and an experiment file in `fog` format.

The host list is a text file in which each line represents the actual network address where a particular virtual host will be instantiated. SherlockFog will connect through SSH to that host and execute the corresponding commands when instructed by the `def` command. Further instructions that involve that node will be executed on the assigned host inside the network container, keeping track of the services, namespaces and cgroups that have been created. Each address in the host list is used only once: If we wanted to instantiate several virtual nodes on the same host, we would have to repeat its address.

The main command line options are as follows:

`experiment.fog` Reads and executes the instructions in file `experiment.fog`.

`-dry-run` Outputs the command list that corresponds to the execution of the `fog` script, but doesn't execute anything.

`-real-host-list file` Reads the host list from `file` (use - for standard input).

`-define key=value` Defines variable `key` to `value` in the main execution context.

`-base-prefix base` Defines the subnet in which the IP addresses of the virtual nodes will be assigned to `base` (in Classless Inter-Domain Routing (CIDR) format).

`-cpu-exclusive` Setup exclusive access to a single CPU core for each virtual host. See section 2.2.2.

`-use-adm-ns` Use an additional subnet for the administrative interface. See section 2.2.4.

`-routing-algo` Defines which algorithm should be used to generate the static routes. The following options are allowed:

- `shortest_path`: Calculates routing rules such that the path from node `A` to node `B` corresponds to the shortest path between them. This algorithm works with every kind of network, but generates a routing rule for *each reachable node*, resulting in huge routing tables if the network is too dense.
- `tree_subnets`: Calculates which subnets are reachable from each outgoing interface and produces routing rules accordingly. It is calculated by iterating through the outgoing interfaces of each node, marking the starting node as unreachable and traversing the rest of the graph using Depth-first Search (DFS), while recording newly reachable subnets at each step. This algorithm produces fewer rules than `shortest_path`, but doesn't allow loops in the network. As there is only a single path between any two nodes if the graph is loop-less, the resulting routing rules are functionally equivalent to using `shortest_path` in this case.
- `world_topo`: Calculates *ad-hoc* routing rules for the World topology that is defined in section 5.5.1.

The complete command line reference can be found in Appendix A.

Fog Scripting Language

The `fog` scripting language defines mainly the following instructions:

- `def vnode`: defines a new virtual node called `vnode`.

  This instruction comprises creating a network namespace with the same name in one of the hosts of the real host pool and adding it to the topology.

- `let var value`: defines a syntactic replacement for expression `{var}` to `value` in the current execution context.

- `for var in start..end..step do cmd_list`: executes `cmd_list` in a loop, binding `{var}` to each value specified by `start..end..step`.

- `include file`: reads and executes every line in `file` in the current execution context.

- `runas vnode user cmd`: executes `cmd` as user `user` in `vnode`.

- `run vnode cmd`: executes `cmd` as `root` in `vnode`.

- `set-delay vnode1 vnode2 delay`: sets link delay between `vnode1` and `vnode2` to `delay`.

- `set-bandwidth vnode1 vnode2 bandwidth`: sets link bandwidth between `vnode1` and `vnode2` to `bandwidth`.

- `connect vnode1 vnode2 delay`: connects `vnode1` to `vnode2`, optionally setting the new link's delay to `delay`.

  This instruction defines new virtual interfaces in `vnode1` and `vnode2`, assigning IP addresses from a newly unassigned P2P subnet to both endpoints.

- `build-network`: this instruction is very important as it defines the routing and ARP tables in every virtual node, using the previously defined topology. The algorithm used to generate these rules depends on the value of the `-routing-algo` option.

  Failure to execute this command will result in the virtual network not being able to route traffic unless a set of rules is defined manually. It also defines the containers' `/etc/hosts` file to be able to resolve the names of every node in the network. Note that every container has the same version of this file, but it must be replicated to be bound to each network namespace, which is done automatically by this command.

The software and configuration requirements are only an SSH server to which a privileged user can connect without a password, and the `iproute` and `tc` toolchains. SherlockFog itself needs not to be deployed to any host but the coordinator, as all connections are established through SSH from there.

The complete language reference can be found in appendix A.

Usage Considerations

We discuss a few usage considerations for SherlockFog:

- As traffic is routed from the host carrier to the right namespace by looking up its destination MAC address, it is not possible to experiment with applications that make use of multicast messages. However, our main focus is on MPI applications and most implementations handle global communication using multiple unicast messages on some sort of virtual tree.

- Total physical bandwidth is shared among nodes. The user must be careful not to overflow the actual carrier. It is possible to avoid this by limiting the maximum bandwidth in each virtual network interface.
- Real link latency must be taken into account when designing the experiment. Since SherlockFog can scale on nodes on different switches, it is likely that pairwise latencies differ. They must be taken into account, as latency is increased on top of the actual link's. As it is the case for all network emulation tools, this could lead to inaccurate results if the desired latency increment is too small with respect to the underlying link's.

2.2.2 Resource Isolation

Network emulators run real code on shared resources. This non-exclusive use may interfere with the performance of the system and, therefore, with the results of the experiments. It has been proposed [60] that careful resource allocation and CPU and network bandwidth limiting could have a positive effect on the quality of a network emulator. As of 2018, the Linux kernel provides six namespaces that function as lightweight process virtualization: `mnt` (filesystem, such as bind mounts and `chroot`), `pid` (remapping of process IDs), `net` (separated instance of the network stack), `ipc` (Interprocess Communication), `uts` (hostname) and `user` (remapping of user IDs). These concepts have been further extended to process groups using cgroups. SherlockFog leverages several isolation facilities in the Linux kernel:

- **Process groups**: A cgroup is instantiated per virtual node. Every process that is executed on that virtual node is grouped into the same cgroup. The use of this technology permits limits to be applied to the whole group as if it was a single unit (see below).
- **Hostname**: For convenience, the SSH server that is used as entry point to instantiate new processes is mapped into a different UTS namespace, exposing a hostname that matches the name of the virtual node.
- **CPU and memory region**: Cgroups allow processes to be controlled for resource limiting, prioritization, accounting, and control. In the case of SherlockFog, this technology is used to limit the CPU bandwidth and memory region that is assigned to a virtual node. CPU bandwidth control is achieved using an API that has been designed to that end for the Completely Fair Scheduler (CFS) [61]. This API allows a fraction of the total CPU time to be allocated to a particular process (or group). The tool also allows a single CPU core to be assigned to a cgroup for *exclusive access*. This option indicates the kernel not to migrate the process group to a different CPU core, reducing virtualization overhead[4].
- **Network links**: Similarly to the case of CPU bandwidth sharing, NetEm is used to limit the total bandwidth that is assigned to a particular virtual link. Oversubscribing the bandwidth of the underlying link is to be avoided, as it may result in network congestion artifacts that change the results of an experiment.

2.2.3 Performance Emulation of IoT Platforms

The CPU bandwidth control mechanism that was described in the previous section is an efficient way to provide an upper utilization bound to that resource and reduce variations in the perceived latency. However, this is not the correct setup to emulate the performance of much slower computing resources, such as those in IoT platforms. This type of platforms usually present different architectures, CPU performance and

[4]This feature is better taken advantage of if there is a single batch process taking up CPU time on that cgroup, such as one MPI process.

features than either HPC or commodity hardware. In order to replay computation accurately, SherlockFog has to support this variation, as it could affect overlapping of communication and computation. Thus, we approach this problem by using an external tool to simulate changes in the compute power of our target CPUs. SherlockFog achieves this by using Intel Pin Tool [19] to slow computation down.

A Pin module, called `slowdown`, has been defined to insert a fixed number of no-op instructions before each machine instruction is executed. The module is implemented by generating, on initialization, code for a function that executes as many no-op instructions as specified via a command line argument. This machine code is clearly platform-specific: we have focused on x86 targets and hence used the `NOP` instruction[5] that is provided by this architecture. This instruction is very lightweight in execution time, allowing the user to fine-tune the delay. The generated function is instructed to be called before each instruction is executed using the `INS_InsertCall` function of the Pin API. The code that generates the sequence of no-op instructions is shown in figure 2.3. This function must be invoked passing a dynamically-allocated memory region in which the function can be generated. The `delay` argument represents the number of times Pin will execute the instruction.

```
void generate_nops(char *a, long long delay)
{
    long long i = 0;
    for (; i < delay; i++) a[i] = 0x90; // NOP
    a[i] = 0xC3; // RET
}
```

Figure 2.3: Dynamic generation of no-op instructions.

As Pin is a JIT compiler, this happens at runtime at the cost of a small execution overhead. The number of instructions is configured per process using a configuration file, allowing platforms with heterogeneous CPU power to be modeled.

Increasing the number of no-ops yields a slower node at a speed that depends on the target platform. A more detailed discussion is found in chapter 4.

2.2.4 Automated Platform Deployment

Some distributed systems have a higher administrative overhead than MPI applications, requiring, for example, configurations to be generated for the virtual network and propagated to every node or services to be initialized in a particular way.

In order to be able to solve this from inside the tool, and since we aim to cover experimentation with many different types of distributed systems, we have provided SherlockFog of a mechanism to deploy, send dynamically-generated client configurations and execute commands via an administrative interface that is connected directly to every container, bypassing the virtual topology. This setup is shown in figure 2.4. The administrative network is implemented by creating an extra container called `adm0` (small dark gray circle above), in which a single network interface of the same name is created and configured in a completely different subnet than the rest of the interfaces. Then, a virtual interface called `adm0` is created on every other container, assigning a different IP address in the administrative network to each of them. This results in every virtual node being connected directly to the rest through a “virtual switch[6]”, which is symbolized by

[5]Single-byte operation; opcode `0x90` [62].

[6]Actually, there is no switch other than the physical one. Traffic in this virtual administrative network is forwarded all the same as in the physical one, the operating systems of each node know to which container a packet should go thanks to macvlan (i.e. by looking up its destination MAC address).

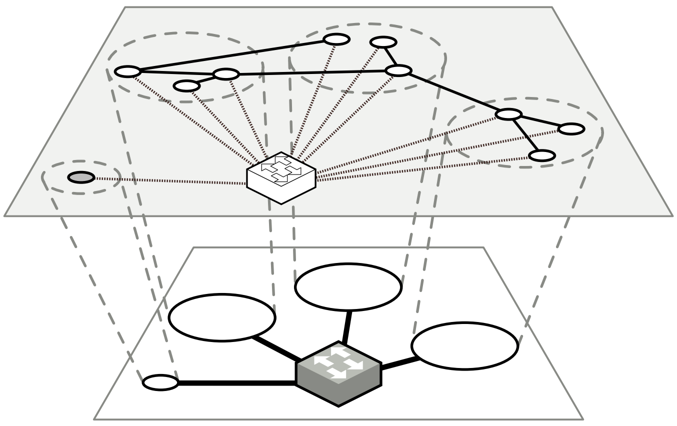


Figure 2.4: View of a sample emulated network that includes the administrative interface links.

the white switch in the figure.

While using this feature, extra care must be taken if launching services that bind to the `INADDR_ANY` address[7], as traffic that is routed through `adm0` is not shaped and reaches every node directly. These additional interfaces can be brought down at runtime by the user if needed.

2.3 Conclusions

In this chapter, we have presented SherlockFog, a tool that uses container-based network emulation to define arbitrary network topologies, in the context of a methodology to study different types of distributed systems following the network emulation paradigm. SherlockFog has few hardware requirements, allowing networks to be instantiated on commodity hardware. It implements a custom language to define the network and execute experiments in a repeatable way.

Its main focus is distributed computing, such as the evaluation of MPI applications, although it is possible to experiment with other types of distributed systems. Since it has been built with distributed computing in mind, it leverages operating system features to allocate CPU resources exclusively to each process. It also includes an external module using the Intel Pin Tool to limit CPU performance to simulate slower IoT-like nodes. This is achieved by dynamically injecting no-operations in the running code. This module may be used from SherlockFog to instantiate an emulated network of slower IoT-like nodes with heterogeneous computing power. Lastly, a mechanism to facilitate starting up services in different virtual nodes in an automated way is described. This is useful to define fully-automated experiments for distributed systems with considerable configuration overhead.

SherlockFog and its associated methodology will be used throughout this thesis to define all experimental scenarios.

[7]The wildcard address that matches every interface, i.e. `0.0.0.0` in IPv4.

3

Validation and Initial Results in Heterogeneous Environments

In this chapter, we explore the use of SherlockFog to study the performance of MPI applications in heterogeneous network topologies. To tackle this problem, we propose validation experiments for our methodology on an application with a well-known communication pattern to show that our tool is able to reproduce different network scenarios faithfully. This analysis corresponds to an application with a communication pattern that is completely sequential, and it is thus possible to describe it using a simple mathematical expression. We follow by analyzing the NPB-MPI benchmarks (see section 1.4) from a communication standpoint. Finally, we evaluate a few custom applications, the HPL LINPACK benchmark and some of the benchmarks of the NPB-MPI set to study the effect of latency in heterogeneous network topologies. The latter are representative examples of scientific code that is implemented using the MPI programming model.

3.1 Related Work

Apart from the tools that we have described in chapter 1, other approaches focus particularly on the use of MPI applications in heterogeneous environments.

Brandfass *et al.* [63] propose a rank-reordering scheme to increase performance of unstructured CFD code on parallel multi-cores. This optimization produces a mapping of MPI processes to CPU cores such that main communication happens within compute nodes, exploiting the fact that intra-node communication is much faster than inter-node in this kind of architectures, using characteristics of the target application. Since load per process is not uniform in unstructured code, it makes sense to reorder processes to reduce frontier communication.

Dichev *et al.* [64] show two novel algorithms for the scatter and gather MPI primitives that improve performance in multi-core heterogeneous platforms. This work focuses on optimizing broadcast trees used by most MPI implementations using a weight function that depends on topology information. However, the user can not experiment using virtual topologies, thus difficulting the study of MPI applications in edge-like environments.

Mercier *et al.* [65] study a more sophisticated process placement policy that takes into account the architecture's memory structure to improve multi-core performance. This proposal is also not suitable for our purposes since the target platform is potentially dynamic and virtual topologies cannot be analyzed.

Navaridas *et al.* [66] study process placement in torus and fat-tree topologies through scheduling simulation using synthetic workloads. This work relies on an execution model which would have to be adapted to study our target platform.

3.2 Validation

In this section, we propose experiments to study the accuracy of our methodology in representing the network scenarios in table 3.1.

The *Barabasi* topology is a random graph generated using the Barábasi-Albert model for scale-free networks with preferential attachment. It represents a connectivity model which is found on the Internet [20]. This topology set was generated using model parameter $m_0 = 2$, then the loops were removed by taking the minimum spanning tree of the resulting graph. Processes are assigned randomly.

The *isles* topologies represent two interconnected clusters of computational resources. These clusters are connected to each other through a single distinguished link. The latency of this link indicates the distance in terms of communication. This scenario represents two sets of nodes in the edge of the network which are connected to a common infrastructure such as the Internet.

Let n be the size of the network, the process placement rules are:

1. The distinguished link connects the first node (node 0) to the last one (node $n-1$).
2. The nodes are partitioned evenly on each cluster.
   - Nodes 0 to $\lfloor \frac{n-1}{2} \rfloor$ go to the first cluster.
   - Nodes $\lceil \frac{n-1}{2} \rceil$ to $n-1$ go to the second cluster.
3. The nodes connected by the distinguished link become the exit nodes for each cluster.
4. Every other node is connected to its respective exit node.

We will show that SherlockFog can emulate different network conditions by analyzing prediction output compared to the expected theoretical results.

3.2.1 Experimental Setup

The experimental platform consisted of six Dell PowerEdge C6145 servers with the following characteristics:

- Four 16-core AMD Opteron 6276 processors each.
- Total RAM per node: 128 GB.
- Operating system: Debian GNU/Linux 7.
- Network interface: 1 Gbps Ethernet.

The nodes have faster and lower-latency InfiniBand connectivity, but it wasn't used as it is not supported by SherlockFog.

The MPI library we have used for the experiments is MPICH 3.2. Open MPI 1.10 was also evaluated, resulting in connectivity issues in some of the emulated topologies, thus favoring MPICH instead. Open MPI uses heuristics to decide if a node corresponds to the local network or not which didn't play well with the network addressing scheme that SherlockFog proposes, i.e. assigning a different subnet per P2P link. This resulted in some nodes failing with *No route to host* messages when starting the ORTE daemons, even though the virtual nodes were actually reachable from one another.

3.2.2 Latency Emulation

In order to show how latency emulation works, we need to use an application with a traffic pattern for which we can obtain an analytical expression for the total communication time. By doing so, we can then compare the expected theoretical time to the output of our tool.

In particular, we have used an implementation of a passing token through a ring logical topology. Each node knows its neighbors and its order in the ring. The token size is configured to be a single integer (4 bytes) throughout this chapter. The number of times the token is received by the initiator (rounds) is also a parameter of the application.

We have analyzed the total number of messages on the network and the execution time for this application, using two different implementations:

- **Token Ring**: implements communication using TCP sockets. This version allows us to have fine grain control of the message generation and protocol.

  It is launched manually by executing it on every host with the following command line arguments:

  - Number of Rounds: how many times the token should return to the first process.
  - Ring Position: a process number, similar to the *rank* in MPI.
  - List of Nodes (in order): the hostnames of each node that forms the ring.

  Every process binds to an IPv4 TCP socket on port 12345 and waits for connections. The first process (ring position 0) connects to its neighbor, which is calculated as a by-one increment modulo the size of the ring and sends the token. As soon as a non-zero-ranked process accepts an incoming connection, it connects to its own neighbor, waits for the token and then forwards it to its neighbor.

- **MPI Token Ring**: the same application, but using MPI for communication. In this case, the objective is to test if the use of the MPI library could also be managed by our tool. In contrast to the TCP version, the only argument is the number of rounds, as the position and node list are handled by MPI directly. It uses the `MPI_Send` and `MPI_Recv` synchronous primitives to send and receive the token respectively.

Since we know the traffic pattern, if we keep the topology unchanged, but increase the latency of one or more links, it would be easy to estimate how much longer the application would take to complete with respect to the original network settings. This increment is calculated as follows: let N be the number of nodes in the topology, t_0 the original execution time, $c_{i,j}$ the total send count from node i to node j and $w_{i,j}$ the shortest path weight[1] from node i to node j, the expected execution time t_e is defined by:

$$t_e = t_0 + \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} c_{i,j} \cdot w_{i,j} \tag{3.1}$$

It is important to emphasize that equation 3.1 represents the expected execution time accurately only since **Token Ring**'s traffic pattern is sequential. Otherwise, we would have to take into account communication overlapping.

We calculated the original execution time, with a fixed latency value on all links, the estimated times for different latency settings and the actual execution times when using SherlockFog with those settings. The full description of the runs is shown in table 3.2.

[1]Each link's weight is set to how much its delay is increased with respect to the underlying platform.

Topology Name	Sizes	Description	Reference in Text
Barabási-Albert	25, 50, 75 and 100 nodes	Random graph generated using the Barabási-Albert model for scale-free networks with preferential attachment.	*barabasi*
Isles	16, 64 and 256 nodes	Two clusters of nodes (star topology) connected through a single path.	*isles*

Table 3.1: Network topologies for validation experiments.

Application	Topologies	Argument Range	Latency	Comments
Token Ring	*barabasi* and *isles* (full set)	100–1000 rounds	10, 90, 170 ms	Latency increased on all edges
MPI Token Ring		100–1000 rounds	5–25 ms (increments of 5 ms)	Latency increased on a single edge

Table 3.2: Parameter configuration for validation experiments

Token Ring

In table 3.3, a partial view of the results is shown. We can observe that the predicted time differs from the measured time by less than 1% in all cases. This is also consistent with the rest of the results for all round counts, latencies, and topologies in our experiment set.

MPI Token Ring

The MPI version produces similar results. In this case, error ranges are slightly higher, but also remain below 1% in all cases, even on a topology on which the logical order of the nodes produces a complex communication path in this application, such as *barabasi*. We consider that this is due to the fact that MPICH handles messaging differently than in our plain TCP implementation, though we find this difference not to be significant.

We can conclude that latency is accurately represented in our tool when executing applications that use MPI for communication on different emulated topologies.

3.3 Communication Pattern Analysis

In order to analyze the communication pattern of each benchmark, we profiled them and recorded for each run a message trace, which included the timestamp, message type, destinations, size, and parameters. The trace was built in a distributed fashion (each node records its own trace). We have used the **mpiP** profiler [58] to generate these traces and compared the results to **simple_profiler**, our own implementation, obtaining a very similar output[2]. Both profilers are implemented by wrapping calls to MPI functions. Instead of linking

[2]The main difference we found between mpiP and simple_profiler was in the timing of function calls. mpiP writes to the log file directly from the wrapper functions, introducing a delay on each function call. Our implementation used another approach, which consisted of recording all function calls in a binary-encoded format in memory, and transforming these records to a text log file at the end (i.e. when `MPI_Finalize` is executed). This produced more accurate results time-wise,

Rounds	Latency (ms)	Predicted (seconds)	Measured (seconds)	Error
100	5.00	206.56	206.52	0.0001
100	15.00	604.50	600.51	0.0066
100	25.00	1002.44	994.49	0.0079
100	35.00	1400.38	1388.45	0.0085
100	45.00	1798.32	1782.46	0.0088
200	5.00	411.28	413.02	0.0042
200	15.00	1203.22	1201.00	0.0018
200	25.00	1995.16	1988.97	0.0030
200	35.00	2787.10	2776.86	0.0036
200	45.00	3579.04	3564.65	0.0040
300	5.00	615.84	619.39	0.0057
300	15.00	1801.78	1810.63	0.0049
300	25.00	2987.72	2983.03	0.0015
300	35.00	4173.66	4177.55	0.0009
300	45.00	5359.60	5347.30	0.0022
400	5.00	820.27	826.30	0.0073
400	15.00	2400.21	2402.16	0.0008
400	25.00	3980.15	3978.04	0.0005
400	35.00	5560.09	5554.26	0.0010
400	45.00	7140.03	7130.86	0.0012
500	5.00	1024.79	1033.60	0.0085
500	15.00	2998.73	3007.53	0.0029
500	25.00	4972.67	4973.75	0.0002
500	35.00	6946.61	6943.70	0.0004
500	45.00	8920.55	8955.64	0.0039

Table 3.3: Validation results for Token Ring, 100–500 rounds, on *barabasi* (100 nodes). Latency added uniformly to all edges.

against MPI directly, they override the symbols of the library, providing for each of them a function that records the event, calls the actual MPI function (recording the call's duration) and returns. This approach requires no changes in the user code.

It was necessary to ensure that clocks were synchronized between nodes to produce consistent, mergeable traces. This synchronization was achieved using Network Time Protocol (NTP) [67] clients on every node. Messages were merged in chronological order to produce a single trace. This final trace was analyzed to produce a communication graph, which has the following characteristics:

- Nodes are numbered by their MPI rank.
- Each pair of nodes n_i and n_j is connected if and only if there is at least one message in the trace that sends data from n_i to n_j, irrespective of the message being P2P or collective.
- Collective messages yield an increment in the message count for every pair involved[3].

although the message features that each profiler output, such as type, size and ordering, remained consistent.

[3]We are interested in the *logical* communication pattern. Even if a collective operation is optimized to reduce message duplication by grouping sends of identical pieces of data that are broadcast to many nodes, we still count one message for each destination.

- Nodes are colored in relation to the closeness centrality property that is calculated using the computed link data, with yellow being the highest and blue the lowest. In all benchmarks except FT, the node with the highest closeness centrality value is node 0. The reason is that this node has links to every other node, as collective messages use it as the root process.
- Node layout has been generated using the SFDP [68] algorithm for undirected graphs. The main criteria of this algorithm are that connected nodes should be drawn closer to each other while representing in the final layout the inherent symmetry of the graph.
- Thicker links correspond to node pairs that connect to each other more times (during the whole run).

These graphs were produced for increasing node sizes to show the structure of the communication pattern configuration as the number of nodes changes.

NPB-MPI has restrictions in the number of nodes that derive from the workload distribution. Most benchmarks require the number of workers to be a power of two, while a few require it to be a squared number. For this analysis, each benchmark has been executed accordingly, following their respective restrictions, from sizes 8 or 9 and up to 64 nodes. As the problem class does not affect the message count and destinations, we show the graphs for class A only.

EP

The communication graphs were produced for node sizes 8, 16, 32 and 64 and are shown in figure 3.1. As expected, the only node that has links to the rest is node 0, which is drawn in the center of each graph.

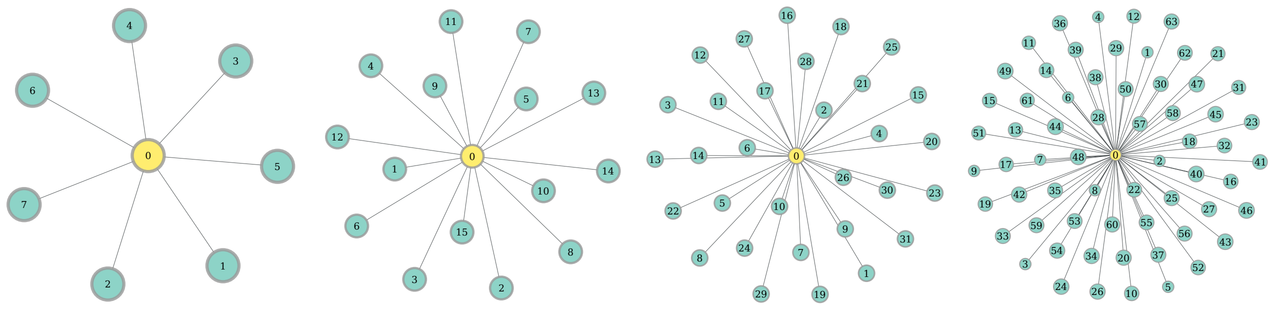


Figure 3.1: Communication graphs for NPB-EP (sizes 8, 16, 32 and 64).

MG

The communication graphs were produced for node sizes 8, 16, 32 and 64 and are shown in figure 3.2. There is a “cubic-like” structure that results from each node having 3 direct neighbors (apart from the root node). Every node is connected to the root node with strong links due to the incidence of the collective messages.

CG

The communication graphs were produced for node sizes 8, 16, 32 and 64 and are shown in figure 3.3. This graphs present structures of 4 to 6 neighboring nodes that are connected to each other following a pattern.

FT

The communication graphs were produced for node sizes 8, 16, 32 and 64 and are shown in figure 3.4. It clearly shows the all-to-all communication features of the benchmark, resulting in all nodes having links to the rest (i.e. a complete graph). All nodes have the same closeness centrality values because of this.

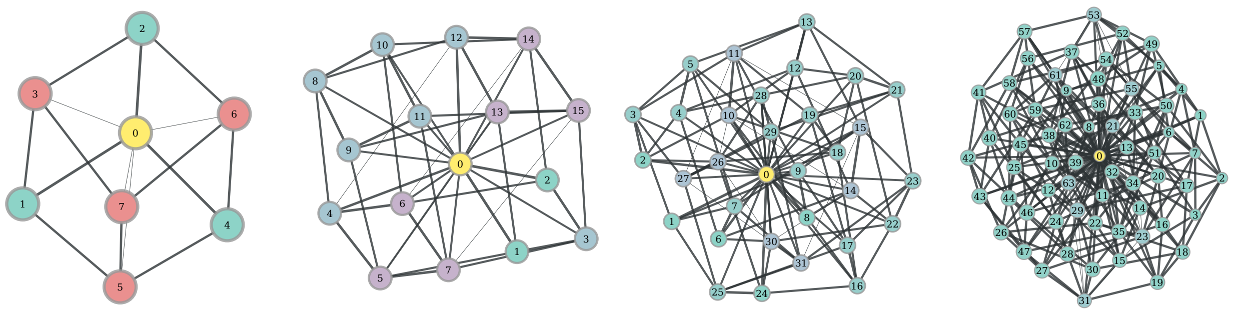


Figure 3.2: Communication graphs for NPB-MG (sizes 8, 16, 32 and 64).

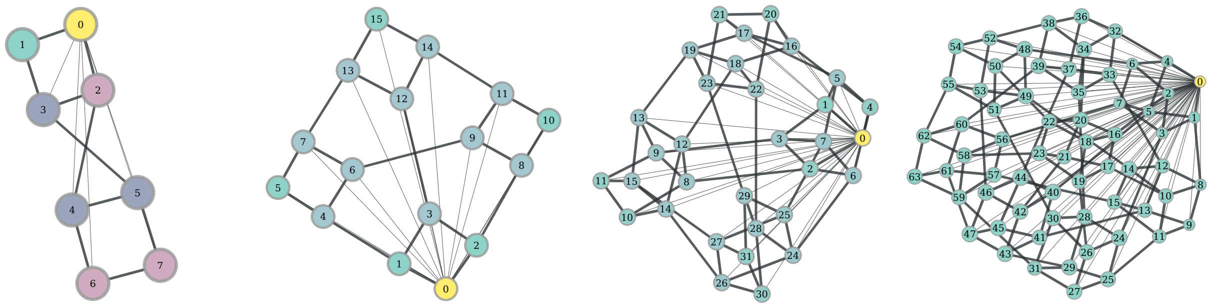


Figure 3.3: Communication graphs for NPB-CG (sizes 8, 16, 32 and 64).

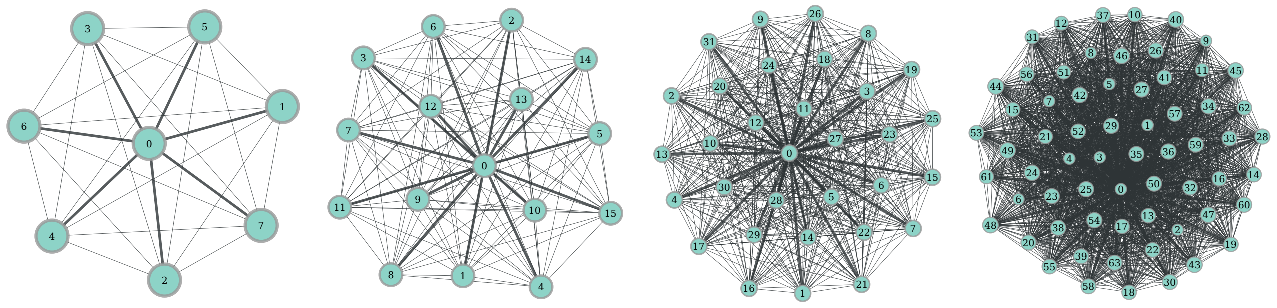


Figure 3.4: Communication graphs for NPB-FT (sizes 8, 16, 32 and 64).

IS

The communication graphs were produced for node sizes 8, 16, 32 and 64 and are shown in figure 3.5. The figure shows a similar pattern to NPB-FT due to all-to-all communication also being present. If this feature is excluded from the trace, we can observe that each node, depending on its rank, has either two neighbors (previous and next in rank, middle nodes only) or a single neighbor (next or previous in rank, first and last only).

LU

The communication graphs were produced for node sizes 8, 16, 32 and 64 and are shown in figure 3.6. The graphs clearly present a grid-like rectangular pattern in which inner nodes have 4 direct neighbors, while edge nodes have either 2 or 3. The root node has links to every other node due to the use of broadcast messages.

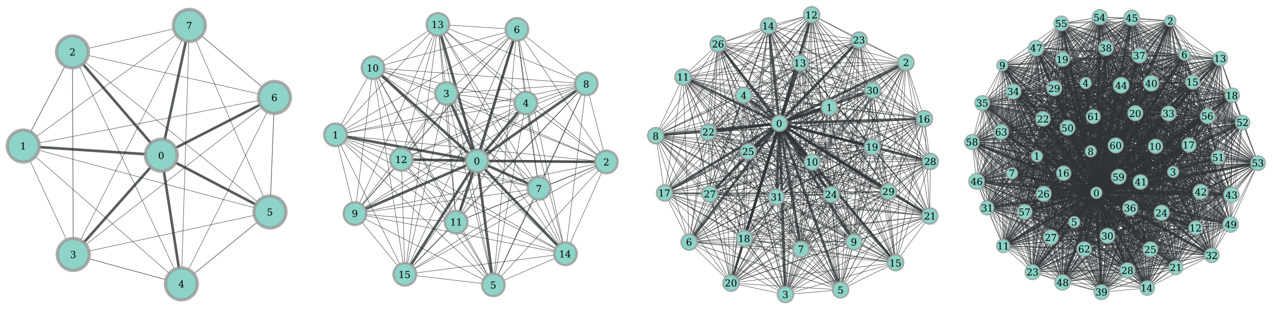


Figure 3.5: Communication graphs for NPB-IS (sizes 8, 16, 32 and 64).

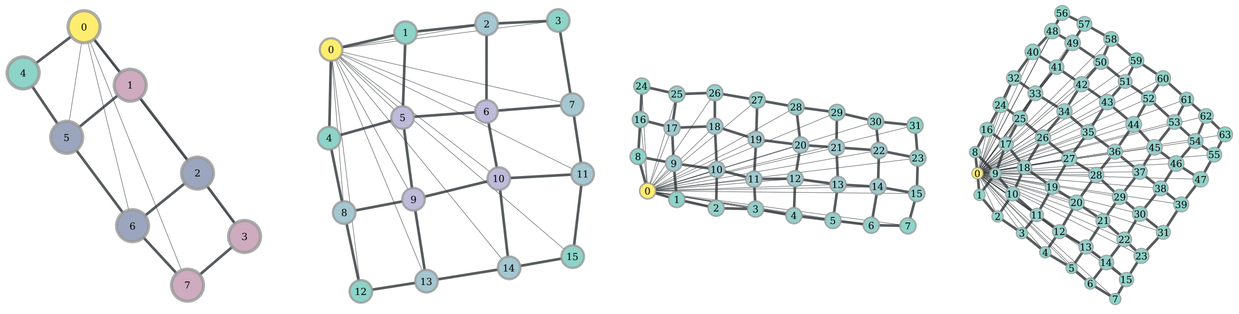


Figure 3.6: Communication graphs for NPB-LU (sizes 8, 16, 32 and 64).

SP

The communication graphs were produced for node sizes 9, 16, 25, 36, 49 and 64 and are shown in figure 3.7. These graphs present a “spherical-like” structure in which every node has a fixed number of neighbors.

BT

The communication graphs were produced for node sizes 9, 16, 25, 36, 49 and 64 and are shown in figure 3.8. This figure is unsurprisingly similar to the SP benchmark, since both solve very similar problems.

3.4 Results and Discussion

In order to evaluate the incidence of latency in MPI applications running in heterogeneous environments, we propose an application set with fixed parameters to be analyzed in a particular network topology, while changing the latency of one or more links. The metric we will analyze is the *slowdown* with respect to base latency values. This is a normalized metric of how much longer it takes for that application to finish if all parameters are unchanged, with the exception of the latency.

The network topologies and sizes used in these experiments are shown in table 3.4. The experiments were repeated 5 times in all cases, producing an average. The semi-transparent patches over the curves in the figures show the standard deviation for each data series.

We will analyze two different application sets: Custom-made and benchmarks. Additionally, a difference will be made between benchmarks, as we have used HPL LINPACK and NPB, which have different characteristics.

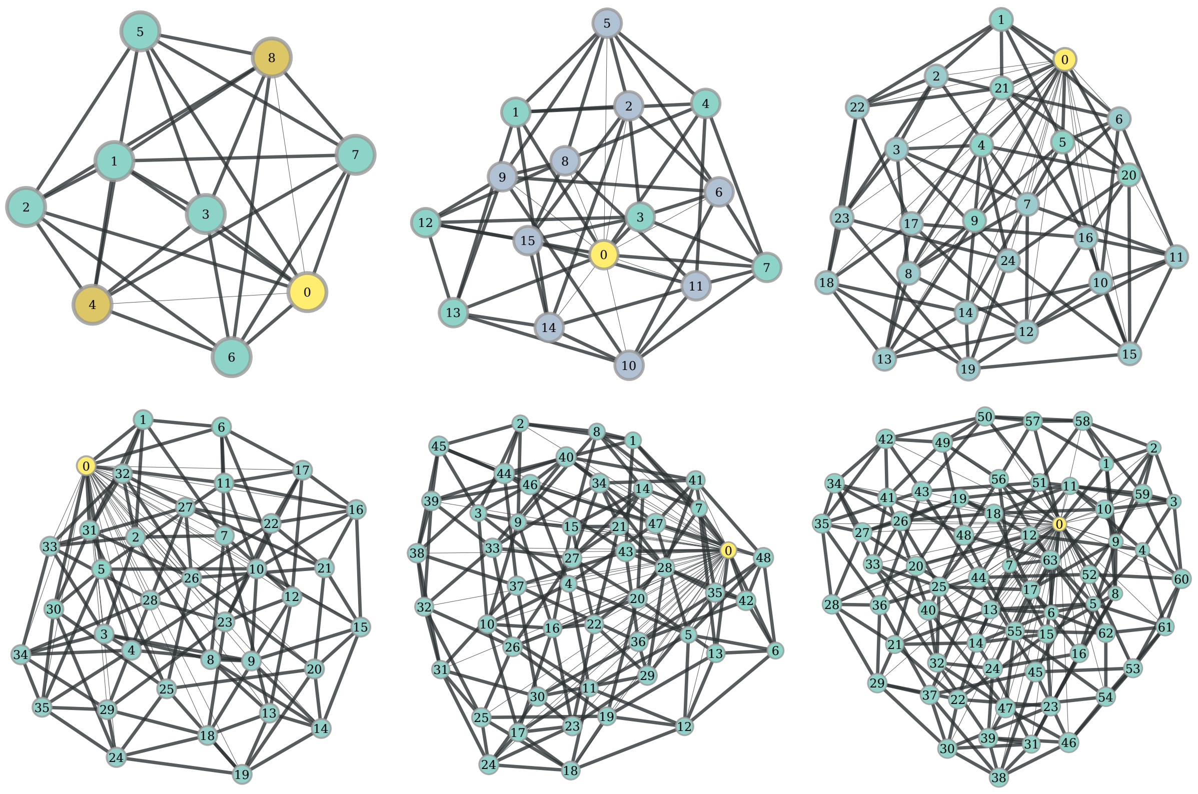


Figure 3.7: Communication graphs for NPB-SP (sizes 9, 16, 25, 36, 49 and 64).

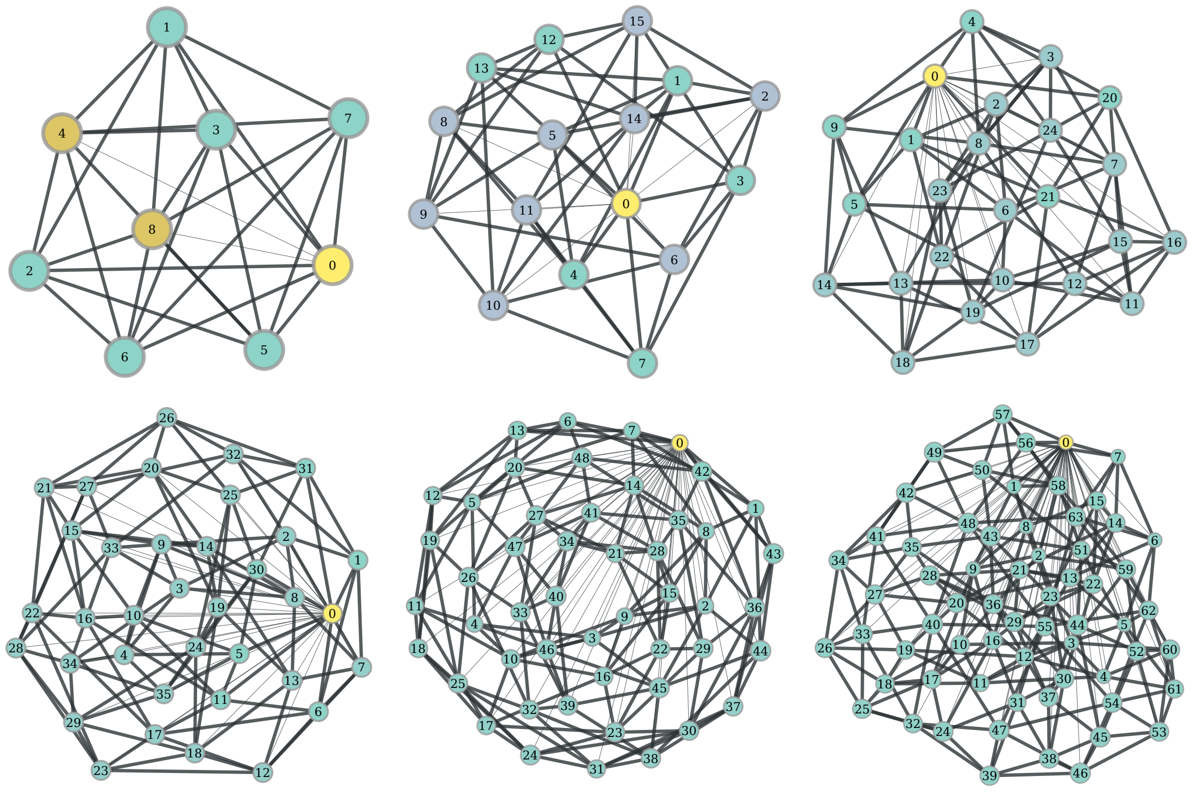


Figure 3.8: Communication graphs for NPB-BT (sizes 9, 16, 25, 36, 49 and 64).

Topology Name	Sizes	Description	Reference in Text
Star	16 nodes	Nodes with single links to a central node (star topology).	*star*
Isles	16, 64 and 256 nodes	Two clusters of nodes (star topology) connected through a single path.	*isles*

Table 3.4: Network topologies for SherlockFog experiments.

3.4.1 Custom-made Applications

The first application that was analyzed is a numerical solution using finite differences for the Navier-Stokes equations in 2-D, parallelized using MPI. We will call this implementation NS2D. We have defined three problem sizes, A, B and C, for grid sizes 100×100, 200×200 and 300×300 respectively. The topologies used in these experiments are a *star* with 16 and 64 nodes.

Latency is increased up to 200 times in all nodes. The slowdown metric for different latencies is shown in figure 3.9. The results for *star*, 16 nodes, size C, are not shown as our implementation didn't work properly in that case.

In the smaller network, the effect of latency is less noticeable in problem size B than size A, with less than a 3x slowdown for 200 times more latency. Increasing the size of the network also results in smaller slowdown values, with size C taking only 1.44 times more to complete when setting the maximum latency.

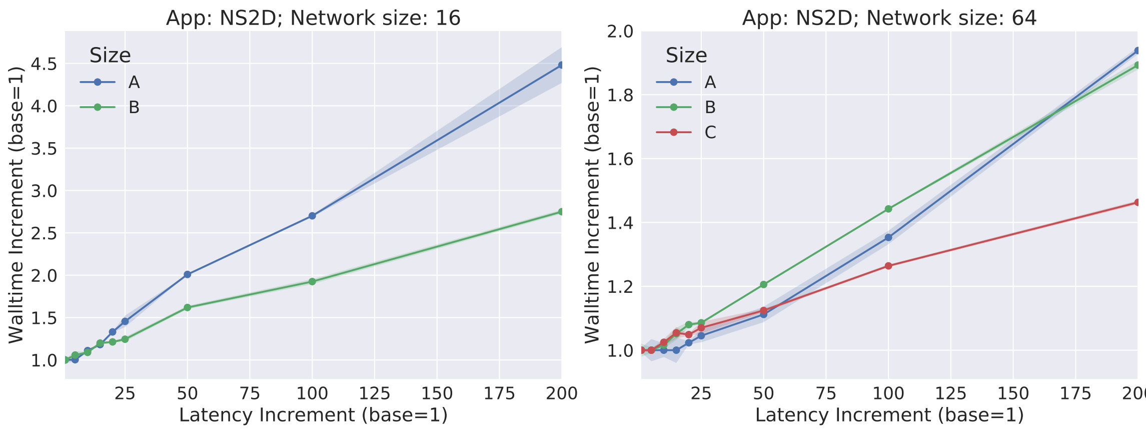


Figure 3.9: Slowdown on the *star* topologies as a function of the increment in the links latency in a Navier-Stokes 2-D simulation.

3.4.2 HPL LINPACK

The LINPACK [69] benchmark is a long-standing evaluation tool for floating point performance, which is determined by solving a dense system of linear equations. The HPL (for Highly Parallel LINPACK) version of the benchmark—actually a benchmark set—is a de-facto standard for raw performance evaluation of HPC systems. It is designed to stress both the processors and the network. Due to this design feature, this benchmark is interesting as it should be a bad fit for heterogeneous networks. Our hypothesis is that the performance should suffer considerably even if the latency of a single link is modified.

Three problem sizes, A, B, and C, are defined for different values of the problem size parameter (`Ns` in the `HPL.dat` file). Each problem size is set to values 5000, 10000 and 15000 respectively. The topologies for these experiments are *isles* with 16 and 256 nodes. The process grid (P and Q parameters) is set as 4×4 for 16 nodes and 16×16 for 256 nodes. The latency of the single distinguished link is increased up to 400 times in the 256-node network and up to 20 times in the 16-node one.

The results for the slowdown metric are shown in figure 3.10. It is clearly seen that the effect of latency is significant for this benchmark, noting particularly for the 256-node network a tenfold slowdown for 400 times more latency in problem size C, and more than a twentyfold slowdown for size A. Slowdown curves are clearly differentiated from one problem size to another. Size A is always affected more than size B, which in turn suffers less of a performance drop than size C. An explanation for this is that each processor has more work to do and is thus less affected by communication latency.

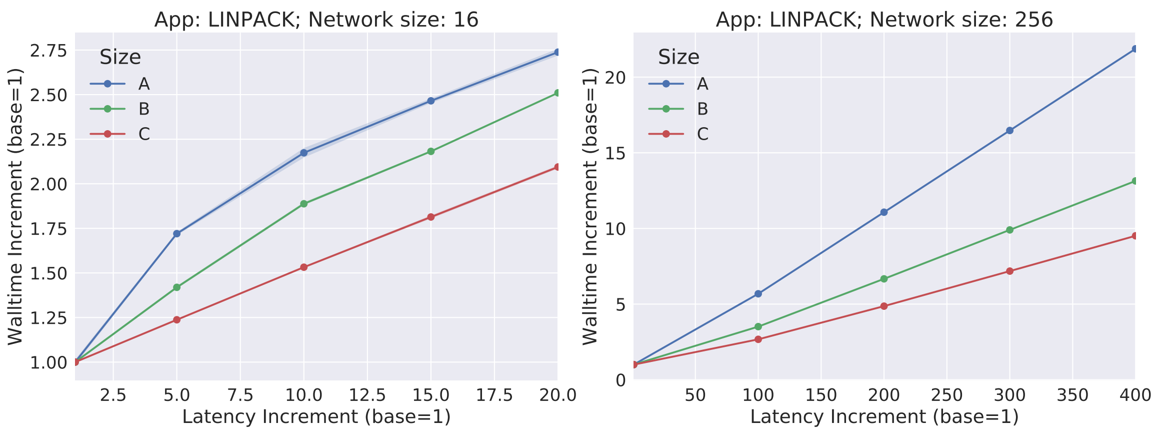


Figure 3.10: Slowdown on the *isles* topologies as a function of the increment in the latency of the distinguished link in LINPACK.

3.4.3 NAS Parallel Benchmarks

In this section, we show the effects of latency on different scenarios in the MPI version of NAS Parallel Benchmarks [21]. These benchmarks were described in section 1.4. We have chosen three of the kernels (IS, CG and MG) and two pseudo-applications (BT and LU) and evaluated performance loss on the *isles* topologies. All benchmarks were executed using SherlockFog to increase the latency of the distinguished link up to 100 times for three different classes (A, B and C).

Our interest lies in finding out how the performance of these benchmarks—which were designed to be executed on a single cluster of nodes with low communication overhead—fares in this use case, comparing total execution time for each latency value to its no-extra-latency counterpart[4].

The results for all benchmarks are shown in figure 3.11. We can observe similar patterns for each network size.

On 16 nodes, the total execution time for all network sizes grows linearly as latency is increased. In the worst case, a hundredfold latency increase results in 14 times slower total execution time. The slope of the curve is usually lower as the problem size grows: class C has less of an impact than the smaller sizes in most cases. We believe this to be related to the fact that each process has more work to do, reducing the impact of the overhead in communication.

On 64 nodes, the difference between classes A and C is more significant. Moreover, we can also observe that the maximum increment is much lower than in the smaller network, up to 3.5 times for a hundredfold latency increase. For BT, CG, and MG (class A), the incidence is also much more significant for smaller latency values. For example, increasing latency 10 times in CG, class A, results in the application taking 2.5 times

[4] In the no extra latency case, the topology remains unchanged, but the latency of every link is exactly the same.

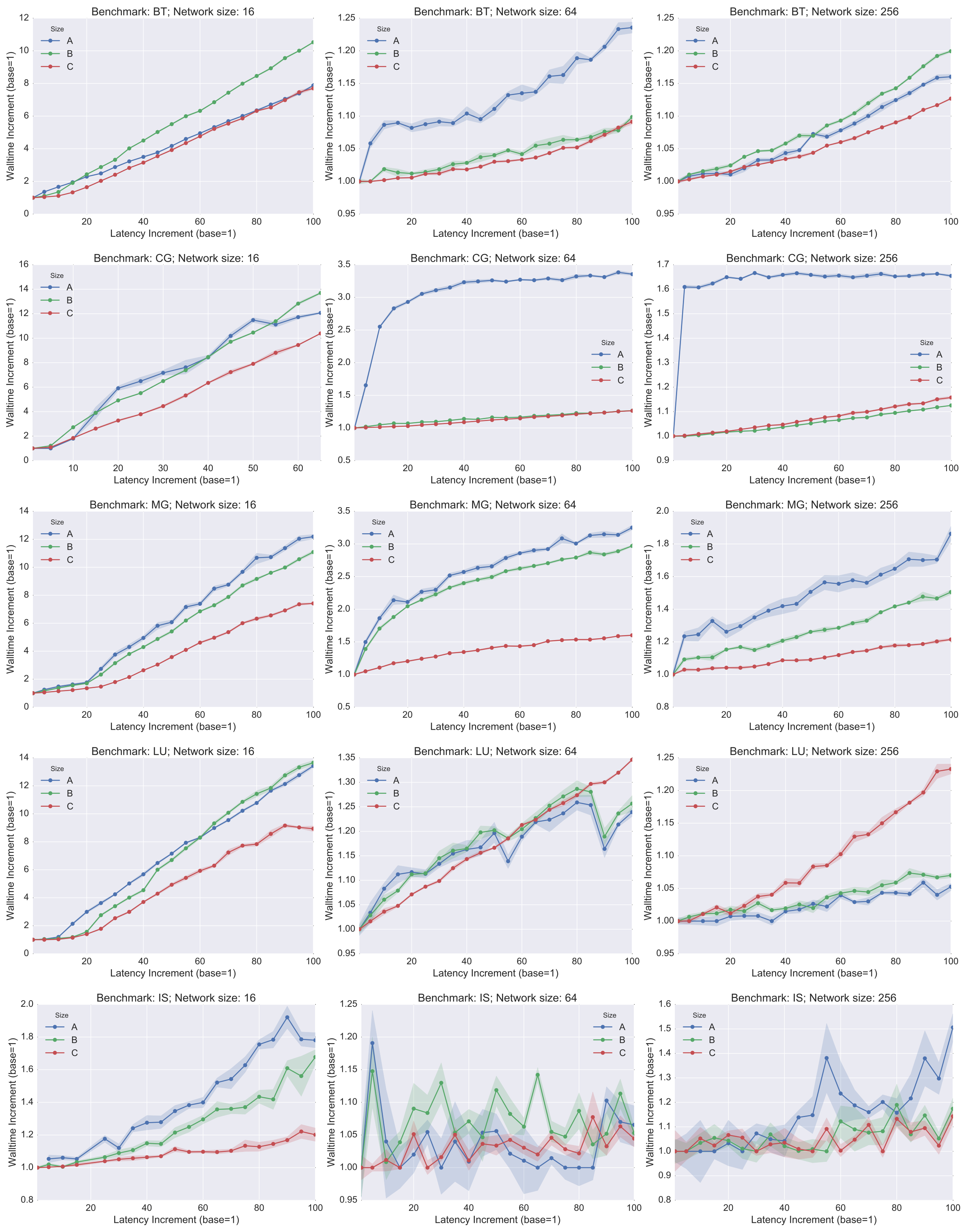


Figure 3.11: Slowdown on the *isles* topologies as a function of the increment in latency of the distinguished link in different NAS Parallel Benchmarks.

more to complete. However, increasing latency 100 times results in it only taking 3.4 times more. LU, on the other hand, doesn't show a significant performance loss for all latency values.

On 256 nodes, we can observe similar results to 64 nodes. However, in this case, the scale is much smaller: the worst case is shown in MG, class A, which takes twice as much time to complete when subject to a hundredfold latency increase. The case of CG is also interesting, as going from no latency to 1 ms results in the benchmark taking 1.6 times more to complete. However, increasing the latency further doesn't produce a noticeable effect. This is similar to the results for 64 nodes, but the effect is more pronounced. We believe this to be related to the communication that goes through the distinguished link representing a much smaller ratio with respect to smaller networks.

Finally, in the case of IS, the increments in total execution time are much less noticeable than in the previous cases. On 16 nodes, the curves for each class tend to drift away from each other as the latency goes up. However, as the node count goes up, the effects of changes in latency on this topology are much less noticeable. We can conclude that this application is not greatly impaired, being the most Fog-ready of all these benchmarks in this particular scenario.

3.5 Conclusions

In this chapter, we have validated the accuracy of SherlockFog in emulating latency in heterogeneous networks.

Latency emulation was validated by estimating the communication overhead in a custom application that implements a token ring. This application describes a sequential communication pattern and is therefore suitable for estimating the overhead theoretically.

We have analyzed a custom-made MPI application, the HPL LINPACK benchmark and five well-known benchmarks that use MPI to reproduce patterns in computation similar to those of CFD applications. We proposed several network topologies in which we have evaluated the impact of increasing the latency of one or more links on the performance of each application.

While some intensive workloads such as LINPACK seem like a bad fit for this type of environments, some results show a linear or sublinear impact, opening up opportunities to use distributed, increasingly ubiquitous computational resources.

4

SherlockFog on IoT Scenarios

In the previous chapter, we have shown that it is feasible to execute certain parallel workloads in Fog/Edge-like environments by analyzing the impact of latency in a particular topology set using SherlockFog. So far, the computing power of the emulated nodes has always been homogeneous. In IoT environments, it is not always the case, as many different types of devices could work together in a single platform. Moreover, traditional HPC nodes have been used to run the experiments, which are usually much faster than IoT devices. In order to be able to answer the question of whether it would also be feasible to use IoT devices for parallel computing, our tool has to be able to model nodes with heterogeneous computing power and IoT-like speeds.

HPC nodes and IoT devices have fundamentally different characteristics in terms of CPU architecture and single-core speed. The instruction set (e.g. vectorized instructions), type and size of CPU caches, speculative execution, and branch prediction are some of the characteristics that have a direct impact on the final performance of a CPU core. It is unreasonable to expect to be able to model another CPU's intricacies without simulation. However, as our interest lies in finding out how a parallel application would fare in an IoT-like scenario, we don't need a complete simulation. It suffices to downgrade the performance of a faster CPU for a particular workload to match the target platform. Many processors have a clock stepping feature that allows them to consume less power, but the number of steps is limited and discretized, making it almost impossible to fine-tune for a specific target platform. SherlockFog implements two mechanisms to emulate slower CPUs that don't depend on the host CPU architecture, each with particular use cases:

1. CPU rate limit using cgroups and the CFS (see section 2.2.2).
2. Use of the `slowdown` Pin module to inject no-operations (see section 2.2.3).

In this chapter, we propose an extension to SherlockFog to model an IoT platform in which MPI applications can be executed. We begin by presenting an IoT platform for network experimentation to which we will compare against. Then, we analyze the CPU rate limit feature and the `slowdown` Pin module to evaluate whether slower platforms are accurately represented when compared to the chosen IoT platform. Finally, we define different scenarios with heterogeneous computing power and analyze them in SherlockFog.

4.1 Related Work

There is no specific work on MPI on IoT platforms, although the literature focuses on smaller deployments of network-connected low-powered devices for execution of parallel applications.

Johnston *et al.* [70] propose the use of several types of Single-Board Computers (SBCs), such as Raspberry Pi, to enable Fog/Edge Computing, IoT and Smart Cities applications. These have some limitations, including limited CPU power, increased hardware failure rate, high network latency, different CPU architecture, and potentially non-uniform hardware. However, they are an ideal platform for education and certain applications such as low-power and memory-limited implementations of neural networks. The authors state that these platforms will help to push the application logic to the edge of the network.

Similarly, Cox *et al.* [71] present Iridis-Pi, a cluster which consists of 64 Raspberry Pi Model B nodes. This platform is an affordable HPC solution for educational purposes. The authors analyze the performance of this architecture using benchmarks, such as LINPACK, and Hadoop for I/O testing, concluding that the lower costs and energy consumption and use of open hardware will increment the number of deployments of this type of clusters.

Mayer *et al.* [72] propose EmuFog, an emulation framework for Fog Computing scenarios. This tool enhances the topology definitions by using a node placement algorithm to find optimal placements of web server replicas in the Internet. While it is possible to use it to evaluate standard Fog Computing applications, building an environment for MPI presents the same difficulties as MaxiNet (see section 3.1), since it is built on top of it.

Although in a different environment, Buchert *et al.* [73] propose a tool called Fracas that handles CPU emulation. This tool is focused on emulating the performance of a multi-core CPU using a load injection tool that takes up CPU time on each core.

Our approach differs from these works as we propose a platform to evaluate unmodified MPI applications on IoT environments, modeling heterogenous network topologies and computing power according to the characteristics of the latter.

4.2 IoT-LAB

In order to validate the capabilities of SherlockFog to emulate IoT-like environments, an experimental platform is required to compare against. We have used the FIT IoT-LAB [25] testbed, which provides access to many IoT devices in geographically distributed sites. This testbed allows the user to make a node reservation and have exclusive access to run experiments. The nodes the user has full access to are the Open Nodes (ONs), low-power devices which are reprogrammed through the serial port when assigned to an experiment. The node's serial port is connected to the Gateway.

Several types of ONs are available. A brief overview follows:

- **WSN430** ON: 16 bit MSP430F1611 micro-controller, a CC2420 or CC1101 radio chip and light and temperature sensors.
- **M3** ON: 32-bit ARM Cortex-M3 micro-controller (STM32F103REY), an AT86RF231 IEEE802.15.4 radio chip and sensors.
- **A8** ON: TI SITARA AM3505 SoC with one 32-bit ARM Cortex-A8 600 MHz mini-computer. Includes 256 MB of RAM, a 32-bit ARM Cortex-M3 micro-controller, an AT86RF231 IEEE802.15.4 radio chip, sensors and a 100 Mbps Ethernet interface.

Additionally, to these types of nodes, control nodes coordinate ONs assignation, reprogramming, initialization, and configuration, while the Gateway is a small Linux computer connected to both the ON and the control node and to the network backbone.

4.2.1 Usage

The platform provides a web interface for management. The user creates an account on the site using their OneLab login. Using these credentials, they can log in to the IoT-LAB web portal. This interface allows the user to load binaries to the devices and provides direct access to the gateways to which the nodes are connected. Additionally, the platform can be accessed through a command line interface. Using either interface, the user can make a reservation of an arbitrary number of nodes of different types in one or several sites to run their experiments.

The command line interface is accessed through the `experiment-cli` command which is provided by IoT-LAB. It allows a reservation to be created and outputs JSON dictionaries with status information of the current experiment and the complete node database. For example, running the following bash script creates a reservation of two A8 nodes in each of the Grenoble and Saclay sites for 60 minutes, then waits for the nodes to be fully initialized:

```bash
RESERVATION_TIME=60
NODES_PER_SITE=2
SITE1=saclay
SITE2=grenoble
experiment-cli submit -d $RESERVATION_TIME \
    -l $NODES_PER_SITE,archi=a8:at86rf231+site=$SITE1 \
    -l $NODES_PER_SITE,archi=a8:at86rf231+site=$SITE2
experiment-cli wait
```

The platform also provides monitoring tools to check the status of the network and measure energy consumption or other sensor information.

Once a reservation has been created, the assigned host list can be accessed by `experiment-cli`, and, in the case of A8, once the nodes are fully initialized, SSH access as superuser is possible from either the Gateway node or other nodes in the same site (using a local IPv4 address) or from anywhere on the Internet (using a globally routable IPv6 address). As expected, nodes in one site can only connect to those in other sites through IPv6 or using a user-provided Virtual Private Network (VPN). SSH access is granted by automatically adding the public key the user provided on the web interface to the authorized keys file inside the node, enabling password-less logins.

In this work, we have only used the A8 nodes, which are the most powerful devices that can be found in IoT-LAB. These nodes are the only ones which are capable of running a complete Linux installation.

4.3 Scenarios and Methodology

We propose a scenario that can be reproduced in IoT-LAB in order to validate the CPU downscaling feature of SherlockFog.

A pair of A8 Open Nodes has been used per experiment. Each pair differs in which sites the nodes are located:

Same site Both nodes are located on the same physical site (Saclay).

Two sites Each node is located on a different site (Saclay and Grenoble).

Recall that IoT-LAB provides a globally routable IP address for each node, but it is reachable using IPv6 addresses only, which unfortunately no MPI implementation to date provided support of. Thus, in order to run this experiment, we had to extend MPICH to support this address family.

This extension comprised the following changes:

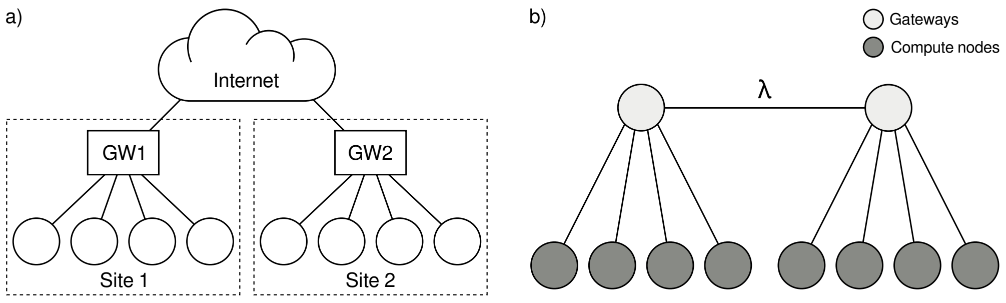


Figure 4.1: FIT IoT-LAB Topology: a) Physical and b) Emulated

- Use IPv6 sockets for all Transport Control Protocol (TCP) communication.
- Use newer IPv6-enabled APIs in name resolution code.
- Modify parameter and host file parsers to accept IPv6 literals, following the format defined in RFC 2732 [26].
- Use IPv6-mapped addresses for IPv4, allowing peers with either IPv4 or IPv6 addresses. This mapping is defined in RFC 4291 [27].

Being able to use IPv6 for communication allowed nodes to connect directly to each other without requiring proxies or IP tunnels that could interfere with communication. The actual topology is shown in figure 4.1.

By measuring latencies on IoT-LAB using ICMP, we have defined the following parameters:

- **Same site** latency: 0.3 ms
- **Two sites** latency: 5 ms

These parameters have been be configured in SherlockFog to match IoT-LAB's induced logical topology when only two nodes are used. The latency of the λ-link shown in figure 4.1 is set to the **two sites** value, while every other link is set to the **same site** value.

Finally, we propose the following methodology to compare against IoT-LAB. IoT platform indicates that the step is executed in IoT-LAB, whereas Experimental platform indicates that it is executed in SherlockFog.

1. Choose an application and runtime parameters.
2. IoT platform $\longrightarrow$ Execute the application on a single node/thread and measure total execution time.
3. IoT platform $\longrightarrow$ Choose two nodes and measure communication latency.
4. IoT platform $\longrightarrow$ Execute the application on two nodes (**same site**) and measure total execution time.
5. Experimental platform $\longrightarrow$ Execute the application on a single node/thread:
    - Run using different settings for the CPU downscaling feature and measure total execution time.
    - Repeat to find the right setting that yields a similar total execution time to the IoT platform.
6. Experimental platform $\longrightarrow$ Configure the CPU downscaling settings as in the previous step to replay the experiment on two nodes using SherlockFog, modelling latency and connectivity as in the real scenario.

7. IoT platform $\longrightarrow$ Execute the application on two nodes (**two sites**) and measure total execution time.

8. Experimental platform $\longrightarrow$ Replay the run in SherlockFog while varying latency parameters to match the IoT scenarios (**same site** and **two sites**).

9. Compare actual total execution time to SherlockFog's predicted values.

4.4 Validation

4.4.1 CPU Rate Limit Evaluation

The Linux kernel allows processes in a cgroup to be limited in the amount of CPU bandwidth that is allocated. The bandwidth is specified using a time allocation *quota* and a time *period*. This limits total CPU consumption to the value specified by *quota* at each *period*. Defaults values for a new cgroup are 100 ms for the period and unrestricted (-1) for the quota. The maximum value for the period is 1 s and the minimum value for the quota is 1 ms.

An initial control experiment with different quota values on the NPB-CG benchmark running on a single core determined that the total running time corresponds to the allocated bandwidth. Using the experimental methodology that was described in the previous section, the validation experiments yielded that the right quota value for NPB-CG, class A, running on IoT-LAB's A8 nodes is 4.02%. The results for this benchmark are shown in figure 4.2.

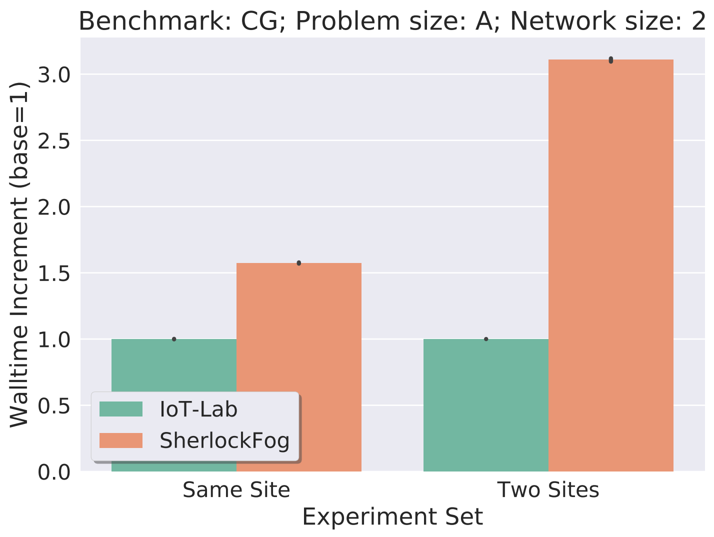


Figure 4.2: Comparison of total execution time between an IoT platform and SherlockFog's prediction using CPU rate limit to emulate a slower platform.

It is clearly observed that the estimation using CPU rate limit does not accurately describe IoT-LAB. The error rate increases as the latency goes up. This shows that, even though the CPU bandwidth is correctely allocated for batch tasks, such as those not requiring communication, the emulation is affected by I/O blocking due to communication.

4.4.2 Intel Pin

The Pin module has a working assumption that the no-op injection measured for a single node/thread (in which no network communication occurs) to be a good estimator for that application using fixed settings. In this case, we have evaluated the NPB-CG (classes S and A) and NPB-MG (class S) using the previously

Benchmark	Platform	Size	Exp. Set	Time (avg)
cg	IoT-Lab	A	Same Site	43.456956
cg	SherlockFog	A	Same Site	47.769897
cg	IoT-Lab	A	Two Sites	48.636216
cg	SherlockFog	A	Two Sites	53.097569
cg	IoT-Lab	S	Same Site	2.249351
cg	SherlockFog	S	Same Site	2.064344
cg	IoT-Lab	S	Two Sites	7.706958
cg	SherlockFog	S	Two Sites	7.644452
mg	IoT-Lab	S	Same Site	0.240845
mg	SherlockFog	S	Same Site	0.189233
mg	IoT-Lab	S	Two Sites	0.505314
mg	SherlockFog	S	Two Sites	0.482823

Table 4.1: Total execution time (in seconds) of an IoT platform and SherlockFog's prediction (NPB-CG class S, NPB-MG classes S and A).

described validation methodology. NPB-MG class A was removed from the experiment set, as it is a memory intensive application, which does not fit in IoT-LAB's limited memory configuration. The parameter of the downscaling mechanism is the number of no-ops that is injected between instructions.

The results of these experiments are shown in figure 4.3. We also show the data (in which the total time has been averaged) that has been used to generate the previous figure in table 4.1.

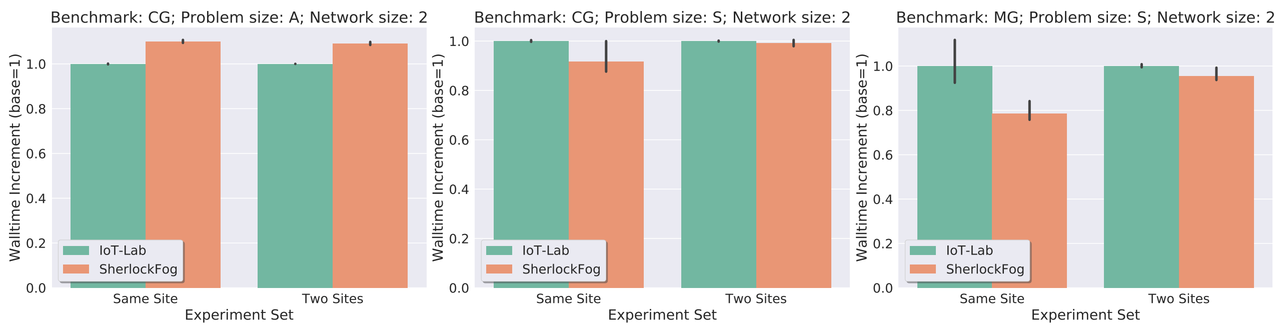


Figure 4.3: Comparison of total execution time between an IoT platform and SherlockFog's prediction using CPU rate limit to emulate a slower platform.

The results show that NPB-CG class A yields an overestimation of 10% with respect to the real platform, although this error rate is consistent even when the latency of the λ-link is increased. This represents a 4 s to 5 s difference in a 50 s run.

In the NPB-CG class S case, the prediction fits the results on the real platform, taking into account that total time in the **Same Site** configuration shows a higher variation in SherlockFog.

The NPB-MG class S case is underestimated by almost 20%. The data shows that the total execution time, in this case, is quite low, which results in a difference of only 0.02 s in total time. This is due to the benchmark taking too little time to finish, which is a factor that has to be taken into account to produce better estimations. This error rate, however, is consistent even when the latency of the λ-link is increased.

These results show that, while not aiming to reproduce a slower platform faithfully, the method provides a good estimator of CPU power that can be used in SherlockFog to analyze the effects of topology changes in heterogeneous environments.

4.5 Experiments

In this section we propose an experiment set to evaluate, using SherlockFog, the NPB-CG and NPB-MG benchmarks on several IoT scenarios with heterogeneous computing power. First, we modeled a network topology in which 16 nodes are equally distributed in two sites, following the previously shown IoT-LAB topology. In order to model this topology, two extra nodes are used. These nodes act as gateways between the two sites and are not used to process computation. A scaled-down version of the resulting topology (six nodes, three nodes per site) is shown in figure 4.1. On the `a)` side, the actual topology found in IoT-LAB. On the `b)` side, the node mapping used in SherlockFog to model this topology. The extra nodes used only for communication are shown in light gray.

4.5.1 Scenarios

Each scenario consists of setting up the CPU power of every node to a fixed amount. We have used two different values, which we will refer to as *slow* and *fast*.

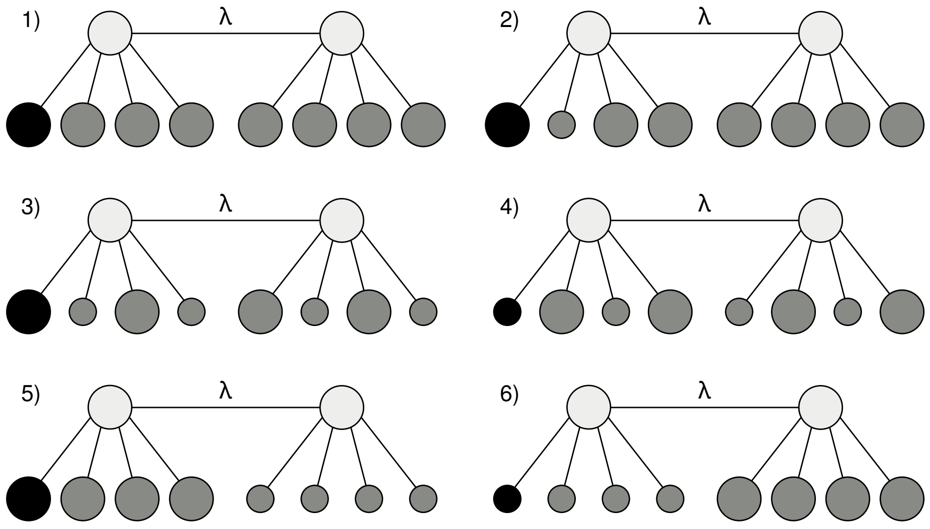


Figure 4.4: Description of experimental scenarios.

These are shown schematically in figure 4.4. Big circles represent faster nodes, while small are the slower ones. The root node is colored in black. Light gray nodes are only used for communication.

The full description is as follows:

1) All nodes are fast.
2) Only one node, non-root and located on the first site, is slow.
3) Neighboring MPI nodes are alternatively fast or slow.
4) Neighboring MPI nodes are alternatively slow or fast.
5) All the nodes on the first site are fast, while the rest is slow.
6) All the nodes on the first site are slow, while the rest is fast.

Both benchmarks were executed on SherlockFog using different latency values for the λ-link, ranging from 0.3 ms to 5 ms, for each scenario.

The results for NPB-CG are shown in figure 4.5. Scenario 1) is used as a baseline to show performance degradation in other heterogeneous scenarios. In the baseline configuration, it is shown that latency affects total execution time linearly.

Scenario 2) shows that slowing down only one node on the first site results in the benchmark taking around 60% more time to finish. Moreover, the effect of latency is reduced significantly. In this case, the difference in node performance makes up for the increased latency.

Scenarios 5) and 6) show similar results as scenario 2), although total execution time is higher.

Scenarios 3) and 4), however, show similar wall time than scenarios 5) and 6), but the slope is similar to the baseline. Alternating fast and slow nodes within neighboring nodes–let us remember that a node is usually connected to its previous and next ranks–yields similar performance to a scenario in which every node is slower.

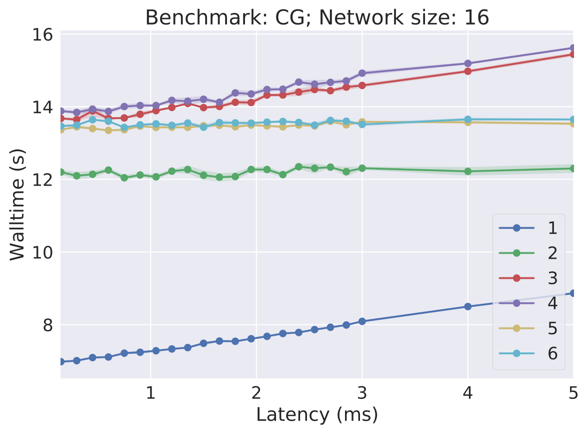


Figure 4.5: Total execution time as a function of latency increments in a two-site scenario; NPB-CG benchmark on SherlockFog.

Results for the NPB-MG benchmark are shown in figure 4.6.

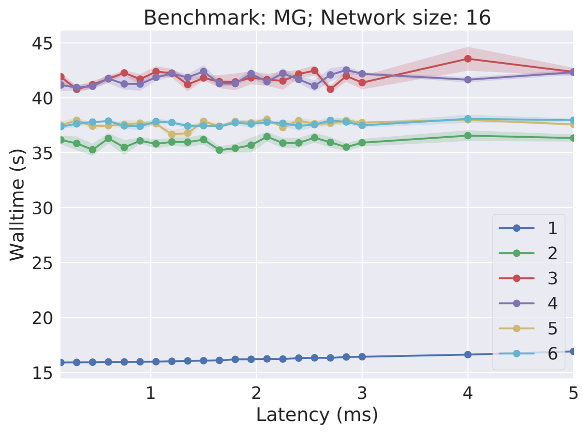


Figure 4.6: Total execution time as a function of latency increments in a two-site scenario; NPB-MG benchmark on SherlockFog.

The results, in this case, show a similar grouping of scenarios.

The fastest is scenario 1), our baseline, while scenario 2) is the second fastest, taking more than twice as much time to complete, which is a bigger slowdown than in the previous benchmark.

Scenarios 5) and 6) come up next, showing a similar slope to scenario 2).

Finally, scenarios 3) and 4) are the slowest ones.

Unlike the previous benchmark, in all cases but the baseline, the latency does not affect performance significantly. This effect is related to the slower nodes making up for the increased latency. An even bigger increment in latency would start showing performance degradation.

It is interesting to note that while degrading CPU power of just one node impacts performance by doubling total time, replacing more of the faster nodes with similarly slower ones does not further reduce performance.

4.6 Conclusions

In this chapter we have evaluated SherlockFog as a tool to analyze the performance of MPI applications in heterogeneous environments, taking into account both CPU power and network topology.

We have evaluated the CPU rate limit and Pin `slowdown` features of SherlockFog to model heterogeneous CPU power. The former is not well-suited to model IoT nodes, but the second one produced estimations agreed acceptably with the experiments on the IoT platform. This mechanism does not aim to reproduce with perfect fidelity different processors and architectures but to provide a way to model performance of much slower IoT-like architectures.

The validation methodology consisted in tuning SherlockFog to capture the slowdown of executing the NPB-CG and NPB-MG benchmarks on the IoT-LAB platform, using a single node. This parameter is used to execute the same application in a two-node topology, using two distinct latency values, which can be contrasted with the real platform. In the case of the Pin module, the results of this experiment show that the estimation is within a 10% error rate unless the total execution time is too small. This error rate is consistent even when the latency of the link is increased.

Then, this tool is used to explore the performance of the same benchmarks on 16 nodes in a particular network topology. We show that degrading the performance of just one node impacts performance globally by doubling or almost doubling total time, but replacing more of the faster nodes with slower ones does not further degrade it. Moreover, the incidence of latency in these configurations is reduced, depending also on the application's communication pattern.

5

Towards a Platform to Study Blockchain-based Distributed Systems

Up to this point, we have been focused on studying scientific computing applications, particularly those that are implemented using MPI. However, SherlockFog has been conceived as a general purpose tool and we believe network emulation techniques represent a viable approach to study other distributed systems.

An interesting family that is currently a hot topic is blockchain-based systems. The most common application of the blockchain technology is to implement distributed, decentralized cryptocurrencies. Creating a new successful cryptocurrency requires an adequate understanding of the consensus algorithms, the structure of the network and the effects the rules have in the stability and scalability of the system. This study is not a trivial task. Moreover, each proposed change to existing deployed currencies is heavily subject to debate by the community, sometimes resulting in network *hard forks*[1] due to the inability to reach consensus among participants. Clearly, under these circumstances, it is important to have some means by which network parameter changes and implementations of new features can be adequately and objectively evaluated in an experimental setup in order to assess their suitability.

In this chapter, we use SherlockFog to propose a methodology to study the Ethereum protocol as a representative example of a blockchain-based cryptocurrency that is based on PoW, while providing a mechanism to reduce the hardware requirements of the testbed without hurting the fidelity of the platform.

5.1 An Introduction to Blockchain-based Cryptocurrencies

Since the creation of Bitcoin in 2008 [74], cryptocurrencies have been gaining traction as a mechanism to enable secure payments without a trusted third-party nor trust between the parties [75, 76, 77]. The technology behind these currencies defines a blockchain, a transaction ledger which is agreed upon different actors in the network through a distributed consensus mechanism. The blockchain technology [78] is also being used to create distributed ledgers in several application fields that exceed cryptocurrencies, such as IoT [79], medicine [80, 81], food traceability [82], insurance policies [83] and others.

This technology, however, has not evolved enough to solve some scalability issues that derive from the high transaction processing and validation times [84]. As with any distributed system, when a new application or consensus protocol is being deployed that is believed to improve performance, it is important to provide a mechanism to evaluate the impact of those changes in the network. Ethereum [85] and other cryptocurrencies use a proof of work to provide trustworthiness to the system, which requires big amounts

[1]A hard fork is a situation in which the network breaks into two or more disjoint subsets of nodes with non-consensual views of the blockchain. See section 5.1.1 for more details.

of computing power to function. This characteristic increases the difficulty of building a testbed to evaluate this type of systems using real implementations.

Recall the automated platform deployment feature of SherlockFog we have presented in section 2.2.4. This feature provides a mechanism to generate and deploy configuration files for every virtual node and initialize any software module that is required for a given distributed system to function on that emulated network. We have used this to implement an automated way to deploy a *testnet* of the Ethereum network on a virtualized infrastructure.

First off, we define Bitcoin and some basic concepts in cryptocurrencies, namely *data structures*, *processes*, and *protocols*. Then, we build on top of those definitions to describe the Ethereum cryptocurrency. We describe the methodology in detail, including which pieces of software had to be modified from the stock installation, and validate it against a real *testnet*. Finally, we propose scenarios to study the properties of the blockchain.

5.1.1 Bitcoin

Bitcoin first appeared in 2008 in a work by an undisclosed party that called themselves Satoshi Nakamoto [74]. It was groundbreaking as it was the first decentralized electronic currency that solved the problem of double spending without requiring a trusted third-party. It is called a cryptocurrency since it uses cryptographic technology—Public Key Infrastructure (PKI) and hashing functions—to ensure the trustfulness of transactions.

A transaction is a codified money transfer that includes information about the sender, the receiver, the amount of money and which transaction was the last one to use that currency. The sender and the receiver are pseudo-anonymous: they are referred to by their wallet's *public keys*, but a single user may possess many different wallets. A wallet is simply a reference to a set of coins that have an owner and which are managed by a piece of software (also called wallet). The owner of that wallet should be the only one that possesses the corresponding private key, which is required to sign a transaction that uses those coins. A sender uses their private key to sign the transaction, which the receiving end can verify to ensure that the former is the actual owner of that money. The reference (hash) of the previous transaction is used to be able to trace from where it comes. The coins that are referenced by a transaction are described by a list of *inputs* and *outputs*. *Inputs* make reference to previous transaction outputs, including its key and the transaction signature, while *outputs* are key pairs that indicate how much of the inputs is transferred and who the destination is. Unused outputs are the only ones that can be used in another transaction and are referred to as Unspent Transaction Outputs (UTXOs). This structure ensures that the problem of double spending doesn't occur: a transaction is invalid if a different one that is already included in the blockchain references the same parent transaction. For a transaction to be correctly verified, the inputs must be validated and the sum of all referenced outputs must be greater or equal to the outputs of that transaction. If it is greater, this exceeding money can be claimed as a transaction fee by the node that processes it.

Transactions don't exist in the system by themselves. They are grouped together into blocks, which also contain additional information in a header, including a reference to the previous (or parent) block. These linked blocks are referred to as the *blockchain*: a replicated data structure that stores the public transaction ledger for the currency. The reference to the previous block is included to ensure that a block cannot be changed afterward without corrupting the blockchain. Since this structure is public, any node can revisit, examine and verify the full transaction history, providing a total disclosure of the operations throughout the life of the system.

A block is the system's basic unit of information and may include zero or more transactions. For transactions to exist in the ledger, they must be bundled into valid blocks and propagated to the network in order to inform the rest of the nodes.

Proof of Work: Creation of New Blocks

Blocks, in addition to the transaction list and hashing key of the previous block, include a field that is called *nonce*, which is used by the mechanism that ensures that some work has to be computed for a combination of valid transactions and a reference to a parent block to be validated. A valid nonce for a block is a number that, when it is hashed together with the rest of the fields, ensures that the resulting value is less than a particular number, which is called the difficulty of the network. Bitcoin uses a hashing function that is based on the SHA-256 algorithm, thus finding this number is only possible by brute-force checking different values. On the other hand, checking whether a block is valid is a very fast operation. A block with the correct nonce is referred to as the *proof of work*, as it requires an amount of computational work to be calculated.

The process of finding this proof that is required to successfully validate and include a new block into the blockchain is called *mining*. Not every node mines new blocks; those that do are called *miners*. There is no restriction for nodes to be miners, but it currently requires a big up-front investment to be able to compete with the rest of the miners and push their own blocks through. If any other node wants to transfer coins, they would prepare the transaction and broadcast it to a miner for inclusion into the next block. In order to promote participation of miner nodes, those that successfully mine a new block receive a reward for that work, apart from the corresponding transaction fees of the transactions that were processed. This reward started at 50 Bitcoins (BTCs) to incentivize early adoption of the technology, but it is planned to decrease as more blocks are mined (every 210,000 new blocks), halving at each point. Since blocks are mined every 10 minutes on average, the halving is expected to occur once every four years. As of October 2018, the block reward is 12.5 BTC (about U$D 80,000 at current rates), with the next halving expected in 2020.

The block reward is the only way in which new money can be created. Total circulation is finite at 21,000,000 coins, which are distributed across time by this reward mechanism.

The miner nodes have the power to decide which transactions of their *mempool*—pending transactions the miner has knowledge of—should go into a particular block. The basis for inclusion could be that the transaction fee is high enough to satisfy the miner. This could drive fees up if the number of miners is low compared to the transaction throughput. Unusually high fees would also impact the usability of the system since it wouldn't be an economically sound decision to generate smaller transactions.

Proof of Work games such as the one used in Bitcoin have statistical properties that ensure that the times in which a new valid block is found follow a Poisson process of a parameter that depends on the difficulty and total hashing power of the network. Since the hashing power can vary due to miner load changes and churn, a difficulty adjustment algorithm is used to drive the expected generation time to converge a fixed number, independently of network changes.

Blockchain: Definitions

The blockchain structure can actually be thought of as a block *tree*, since more than one block may reference the same parent. In this context, a chain of blocks is any path in the tree that goes from a particular node to the root. The chain that has the biggest total difficulty, which is calculated as the sum of the difficulties of every block, is called the *main chain*. Blocks not in the main chain are referred to as *stale*. It is a desirable property of the system that the number of stale blocks is as low as possible since they represent wasted miner work. A related concept is that of *orphan blocks*, which corresponds to mined blocks whose parent is not valid, rendering them impossible to validate.

The root of the blockchain is a special block called *genesis block*. It is the oldest, dating from January 3rd, 2009. It includes a piece of news from The Times[2] in the *coinbase* parameter, to prove that it hadn't been mined previously. The block number corresponds to its height in the tree, with the genesis block being #0.

[2]Incidentally, it is a title from the Economy section: *The Times 03/Jan/2009 Chancellor on brink of second bailout for banks*.

Establishing Consensus

Due to the distributed nature of Bitcoin, it is possible that two or more miners find different valid blocks with the same parent (i.e. the same block number). In this situation, these blocks are said to be *competing*. As blocks are propagated to the rest of the network, a node might find that it is mining a new block with a parent that is different from the block it has just received. This requires a mechanism to decide which one is the good block: this mechanism is called the *consensus protocol*. In Bitcoin, the block that wins is the one that makes the total difficulty of the branch as great as possible, using the first-seen rule on draws.

A situation in which one or more nodes have a different view of the blockchain is called a *fork*. The system is said not to be under consensus if there is a fork. Eventually, the consensus protocol should be able to resolve forks in order for the system to function properly.

There is also another type of forks that are not strictly technical in nature. Implementing changes in the Bitcoin protocol requires every node to upgrade to a new version. This generates heated discussions in the community in order to adequately justify the proposed changes. If at least one node keeps the old version, the set of rules it would use to maintain the blockchain would be different from the rest of the nodes, resulting in two incompatible networks. This situation is called a *hard fork*.

Bitcoin already has a history of hard forks that sprung other cryptocurrencies such as Bitcoin Cash and Bitcoin Gold.

5.1.2 Ethereum

Ethereum builds upon the ideas of Bitcoin, proposing improvements in terms of transaction throughput and including the use of Smart Contracts to enable the specification of more complex payment conditions [86]. In terms of the efficiency of the protocol, the most salient difference with respect to Bitcoin is that the expected time between blocks is reduced considerably: from 10 minutes to 14 seconds on average. The hashing algorithm is also different, as it uses SHA-3 and other techniques that make it explicitly difficult to efficiently implement in hardware. This renders existing deployments of mining hardware for Bitcoin useless to mine Ethereum.

Consensus in Ethereum

As the average block generation time is much lower than in Bitcoin, Ethereum implements a simplified version of the Greedy Heaviest Observed Subtree (GHOST) consensus protocol [87]. The main motivation behind this is that fast confirmation times generate a high number of stale blocks, which affects the security of the system. It also affects centrality: if miner *A* has 30% of the hashing power and miner *B* has 10%, *A* will have a risk of producing a stale block 70% of the time (i.e. every time it is not *A*'s block the one that ends up being the latest in the chain), while *B* will have a 90% chance. Thus, if the block interval time is short enough, *A* will produce many more blocks that end in the main chain simply due to its power, a situation that favors these big players in detriment of the smaller miners.

The GHOST protocol proposes to include stale blocks in the calculation of which chain is the longest. The stale descendants of a block's parent are called *uncle* blocks. Up to two of them can be included in the calculation of the most difficult chain. The issue of the centralization bias is solved by providing block rewards to stale blocks: a stale block receives 93.75% of the base reward, while the nephew that references the stale block receives the rest. Transaction fees are not awarded to uncles. Ethereum implements a simplified version of this protocol, limiting uncles valid for inclusion to a maximum of seven levels[3].

The calculation of the chain difficulty in Ethereum is similar to Bitcoin: the total difficulty is the result of adding the difficulty of each block, ignoring referenced uncle blocks. Ties are resolved by choosing the

[3]The implementation details of this features are more extensively described in the Ethereum whitepaper that is found at `https://github.com/ethereum/wiki/wiki/White-Paper`.

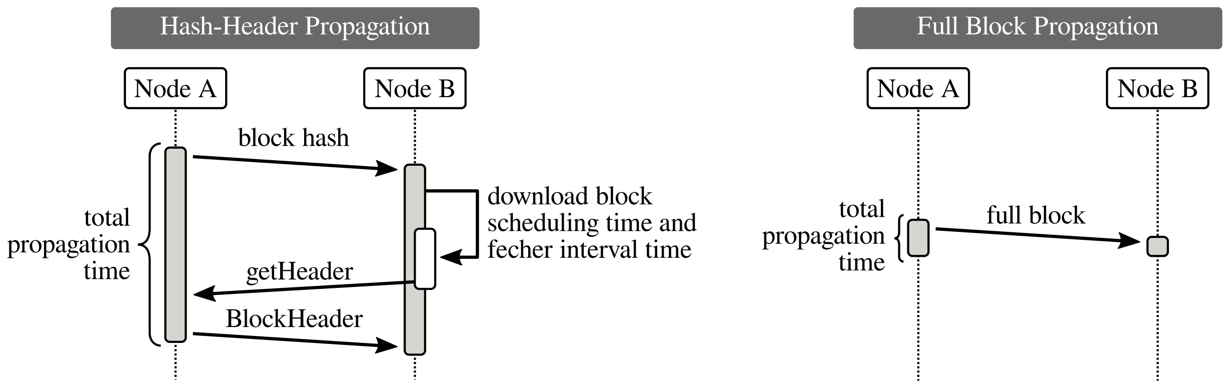


Figure 5.1: Message diagrams for deferred (*hash-header*) and full block propagations in Ethereum.

winner chain using a random heuristic [88].

Transactions

The transactions are specified by a piece of code or Smart Contract, which is executed by a virtual machine on each node to process and validate it. This contract has some limitations in how many instructions can be executed and the amount of memory that can be used, in order to avoid taking up too many resources. The cost of a transaction is expressed in terms of an amount of *gas*. The `GasPrice` field provides a conversion from gas to Ether, while `GasLimit` specifies a gas limit. Similarly to Bitcoin, the senders and receivers are described using the public key of their wallets.

Protocol

Ethereum defines a P2P protocol that is used to propagate new blocks and communicate pending transactions throughout the network. We discuss some characteristics of the communication protocol by first presenting the main messages:

`Status` Handshake message, informs protocol version.

`NewBlockMessage` Sends a new block.

`NewBlockHashes` Sends only the hashes of new blocks.

`GetBlockHeaders` Requests the headers of one or more blocks.

`BlockHeaders` Response to `GetBlockHeaders`.

`GetBlockBodies` Requests full blocks.

`BlockBodies` Response to `GetBlockBodies`.

The block propagation protocol is used by the miners to communicate new blocks to the rest of the network. Non-miner nodes request unknown hashes or full blocks to neighboring nodes. If only the hash is being sent, it is said that it is a deferred download using a *hash-header propagation*, whereas unsolicited full blocks are referred to as *full block propagations*. Both mechanisms are described in figure 5.1.

Every client keeps track of which blocks are known by its neighbors in order to reduce the number of messages that are required to propagate the blockchain.

5.2 Related Work

Additionally to existing efforts of general-purpose network emulation and simulation tools that have already been covered in the previous chapters, we discuss blockchain-focused approaches.

On the simulation front, Miller and Hopper present Shadow [89], a discrete event simulator which makes use of direct code execution to run multiple instances of the Tor client in a simulated environment. Using function interposition, it replaces network and time operations with calls to the simulator. Shadow has some limitations, such as the fact that it is unable to execute a multithreaded code. An extension called Shadow-Bitcoin [90] solves this limitation by using a user-level thread library, enabling the execution of the `bitcoind` mainline Bitcoin client. This solution requires an engineering effort to be adapted to other cryptocurrencies or distributed systems. Moreover, even though the time dilation strategy provides faster-than-real-time execution for small networks, it becomes slower as the size of the network grows, since it uses a single thread to simulate the complete network. The authors cite an up to 40x slowdown (compared to real-time) on server-grade hardware when simulating a 5,760-node network.

Gervais *et al.* [91] analyze the impact on the security of the Bitcoin network of reducing the target time, using a simulator built on top of NS-3 [39].

There are also several efforts that are based on the experimentation approach.

Decker and Wattenhofer [92] use an instrumented client to analyze the block propagation time and fork generation on the Bitcoin network. Their client connects to a big amount of nodes, collecting information but without forwarding it to other nodes. They have collected information during the generation of 30000 new blocks and studied the blocks mean propagation time and the fork generation frequency as a function of the propagation time, concluding that there is a correlation between the propagation time and the size of the blocks.

Miller *et al.* [90] define a new tool called *Coinscope*, whose goal is to improve the characterization of the distribution of nodes on the Bitcoin network. The authors study, by partitioning transactions, which nodes are more *influential* on the network. Besides, they estimate how many peers a Bitcoin client has on average, concluding that it is close to the default value of the official client (eight peers). However, they have also found nodes with more than 100 peers each, which is attributed to the existence of *mining pools*.

Our work builds on top of the network emulation platform, based on MaxiNet, that was defined by Vanotti [93] and Vileriño [94] to study the Bitcoin and Ethereum networks respectively. Vanotti defines the Bitcoin client modifications, blockchain profiling methodology, and a scaled-down world network, whereas Vileriño extends this work to the Ethereum network. The work that is presented in this chapter eliminates a scalability problem of that platform that is related to the use of a static controller to route traffic in MaxiNet.

5.3 Changes to the Ethereum Reference Client

We have used the `geth` Ethereum client as a reference implementation to show the potential of our methodology. The mining process has been modified to produce a similar block generation pattern without the costly computation involved. This client has also been instrumented to record network events at each node for offline processing. The messages that were instrumented are those described in section 5.1.2.

5.3.1 Simulated Mining

In Proof-of-Work-based systems such as the Ethereum network, the miners use their own CPU power to find new candidate blocks for the blockchain. If we wanted to emulate the whole Ethereum network, which consists of thousands of nodes, this would require unattainable amounts of computing power on the platform nodes. The emulation capacities of the system are also limited by the total hashing power that is available on them.

We propose a solution that takes into account the characteristics of the mining process to provide input for the whole system that doesn't require computing power. In Proof-of-Work systems, the difficulty of the network is dynamically adjusted such that the combined hashing power of every miner in the network produces on average a single block every t units of time. The value of t, usually in the scale of seconds or minutes, is called the *target time* of the network.

Recall from section 5.1.1 that the generation of new blocks follows a Poisson process such that the expected time for a new block to be generated converges to the *target time*. We have used this information to model the mining process. It works as follows: every miner throws a weighted coin to decide whether they have to mine a new block or not. The probability of the coin depends on the hashing power of that miner. Since it is not necessarily true that the hash value of this new block has any given property that represents the proof of work, we can say that it is not valid in the sense of the Ethereum protocol. Thus, the validation step is removed from every node, taking into account that it takes a nontrivial fixed amount of time to complete, which is included in the simulated protocol. The consensus mechanism works in the same fashion as in the real system, forwarding incoming blocks as if they were valid but without checking them.

5.3.2 Event Instrumentation

In order to produce a global view of the blockchain, each client has been modified to generate a *log file* in which all relevant network events are recorded. This information is stored with a timestamp that indicates when the event was processed by the client.

To ensure that the events of the combined log files could be processed as a whole, the clocks in the network have to tell the same time. NTP clients are executed on the non-emulated network, which is not traffic shaped, to keep the clocks of the hardware testbed synchronized at sub-millisecond precision. This mechanism proved to be precise enough, as the number of events that were recorded at the source with a timestamp that lies more in the future than its corresponding event at the destination represents less than 0.1% of all events, even in the most severe network conditions we have used in our experiments.

5.4 Fork Metrics

Building a blockchain is all about establishing consensus in a decentralized way. Ideally, the consensus protocol should resolve competing blocks while allowing every node to be informed. A situation to avoid is that of forks: two or more nodes having different *views* of the blockchain. Several characteristics of the network could have an impact on the generation of forks, such as the generation delay of new blocks and the topology of the network.

We defined three metrics to study the performance of our private network with respect to the generation of forks.

5.4.1 Orphaned Block Rate

The orphaned block rate is defined by counting the blocks that have been mined by any of the miners but do not end up in the main chain. This count is normalized by the total number of mined blocks. It gives an insight on how much computation has been wasted on the network to produce the final blockchain.

Rate of blocks with multiple descendants

This metric is defined by counting the blocks that are being pointed to by more than one block, normalized by the total number of blocks in the main chain. If the network was completely fork-less, every block would be pointed to by 0 or 1 blocks and this metric would give out a 0. Instead, its value will increase with the number of branches in the main chain, irrespective of the depth of those branches.

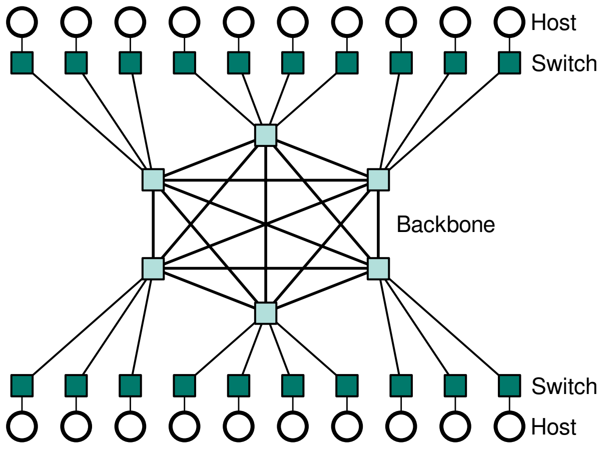


Figure 5.2: Schematic view of the World Topology. The backbone models inter-country links, while the Host-Switch pairs connected to a particular node in the backbone correspond to nodes in the same country.

5.4.2 Deviation from Target Time

A well-functioning blockchain should produce a new block every *target time* units of time on average. However, if some of the blocks do not end up in the main chain due to forks, it means the amount of time that is required for a block to actually reach the main chain is higher than the target time, as more blocks have to be mined. This metric represents that notion. It is calculated by dividing the aggregated block generation times to the number of blocks in the main chain, then normalizing this value to the target time of the network.

5.5 Network Topologies

We have to distinguish two different kinds of topologies: physical and logical. The physical topology refers to the underlying network that is used to route traffic. The logical topology is related to the connection pattern being followed by the application.

Two different topology types have been defined for the physical network: *World* and *Fully Connected*.

5.5.1 World

In this topology, every node has a tag that indicates from which country it is. Each node is connected to a single switch, which is in turn connected to the network backbone. The backbone is defined as a clique of switches, each of them representing the exit node for a particular country. The routing strategy is defined such that every two nodes that are tagged in the same country use their country's exit node to connect to each other, while two nodes in different countries use the link that connects the exit nodes of their respective countries. This is shown schematically in figure 5.2.

The number of nodes assigned to each country is proportional to the number of clients of the Ethereum network in that country as is published in `ethernodes.org` [28].

The latency model was generated using a database of average latencies between countries. The latency of a link that connects two backbone switches was defined as half the latency between those countries. The link that connects a node's internal switch to its corresponding backbone switch is defined to have half of the intra-country's latency that is recorded in the dataset. This model has limitations as some of the countries that have the highest number of nodes also have big variations in their intra-country latency due to their geographical extension (e.g. China, Russia, USA). However, this could be improved by using finer-grained information of the latency distribution of the network.

5.5.2 Fully Connected

This topology is a K_N graph, with every link being shaped to the same latency (1 ms).

The routing strategy is such that every node is reachable in exactly one hop (using the link that connects directly to it).

5.5.3 Logical Topology

The logical network refers to the neighbor information each client has. Two nodes are connected in the logical network if there is an established connection between them. Note that it is not required for the nodes to be close in the physical network, potentially requiring a long traversal to reach each other.

The logical network has been defined randomly for each topology type and size. Client nodes are statically configured to keep a neighbor list of 2 or 3 randomly chosen nodes. Node discovery has been disabled.

In all configurations, given a network of size N, $\frac{N}{2}$ randomly selected nodes are miners.

5.6 Validation

First off, we will compare the block generation events using simulated mining on SherlockFog to a real scenario on a private Ethereum network.

For the real scenario, we have used a computer lab consisting of 20 Core i5-3550 CPUs with 8 GB of RAM, each running at 3.2 GHz, and connected to a 100 Mbps Ethernet switch. In this case, an Ethereum testnet was set up with our modified client, with simulated mining disabled and a custom target time of 21 seconds. Half of the nodes were configured as miners, using a single CPU core for that operation.

The simulated mining scenario was executed on a Core i7-3370 CPU with 16 GB of RAM, running at 3.4 GHz. The target time was also set to 21 seconds. The physical topology is a K_{20} graph (Fully Connected). Each link has a delay of 0.1 ms. This delay is half the latency of that of any pair of nodes in the real scenario, as measured by ICMP. The same logical topology has been used in both scenarios.

Let t_n and t_{n+1} be the points in time in which blocks n and $n+1$ have been mined, we have recorded the block generation delays $t_{n+1} - t_n$ of the first 1000 blocks that actually reached the main chain in both scenarios.

Figure 5.3 shows that the generation delays in both scenarios follow an exponential distribution a similar mean. Moreover, the fork rate is similar in both cases as well.

We can conclude that simulated mining on a K_N topology behaves statistically in a similar way to the real system running on a local network.

5.7 Blockchain Evaluation

The experimental setup used in this work consisted of 6 Core i7-2600 desktop workstations with 8 GB of RAM and a stock version of Ubuntu 16.04 LTS. The instrumented client is `geth` v1.5.8.

Using our framework, we will analyze the incidence of the target time on the metrics we have defined in section 5.4 for the topologies World (sizes 50, 100 and 200 nodes) and Fully Connected (sizes 50 and 100 nodes).

5.8 Results and Discussion

In our first experiment, we analyze the orphan block rate as defined in section 5.4.1. Figure 5.4 shows the results for World and Fully Connected topologies. As the target time increases, the orphaned block rate

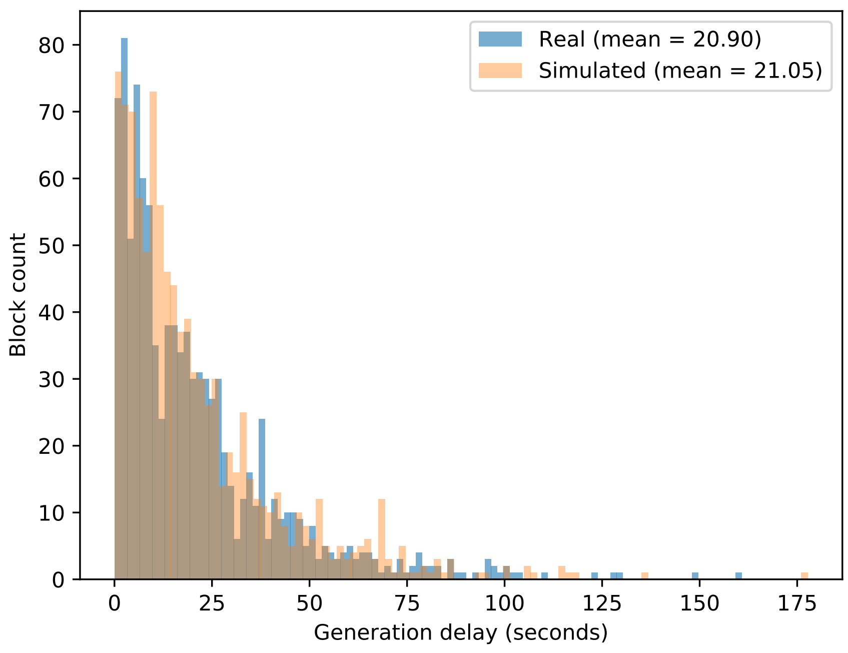
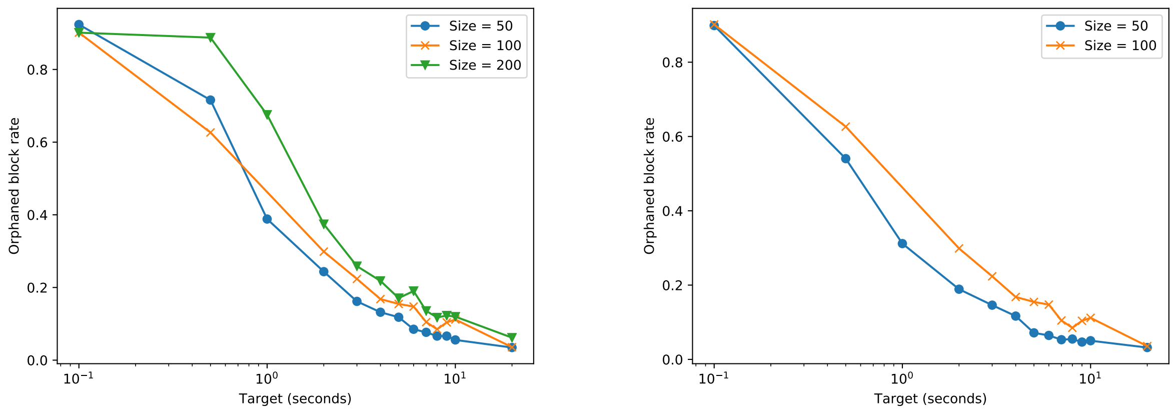


Figure 5.3: Distribution of block generation delays

decreases to reach values lower than 0.1 s for 10 s and higher. This behavior is similar in all network sizes, although rates are slightly higher for bigger network sizes at the same target time. Both topologies behave similarly, although Fully Connected produces fewer orphan blocks at lower target times than World, as blocks are being able to propagate faster throughout the network because of the very reduced paths towards the target nodes. Nevertheless, it is clear that target times lower than 1 s are too fast for either of the network topologies, as orphan blocks represent 90 to 40% in those configurations.



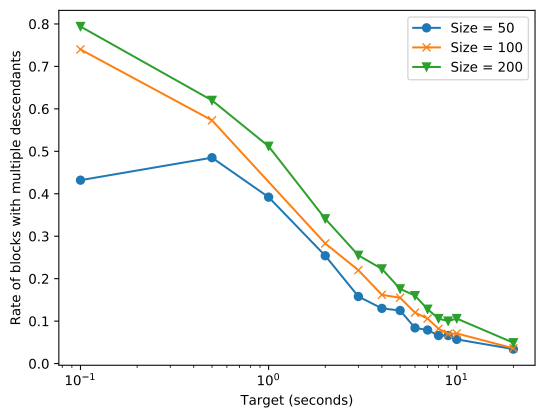
a World Topology

b Fully Connected Topology

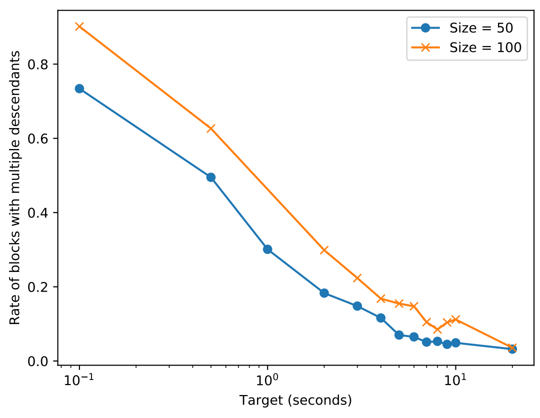
Figure 5.4: Orphaned block rate under different working conditions varying the network size.

Figure 5.5 shows the results for *Rate of blocks with multiple descendants* (defined in section 5.4.1). Similarly to the previous metric, this rate decreases as the target time and the size of the network increase. Again, an interesting case is target time 0.1 s, in which the value of the metric increases at a lower rate (or even decreases in the case of World, size 50) with respect to the target value which is immediately above. This effect is due to branching occurring further down other branches instead of directly on the main chain, thus not being accounted by this metric.

Finally, figure 5.6 shows the results for the metric defined in section 5.4.2. It is interesting to observe that low target times require up to 10 times more time to produce a single block in the main chain. In the



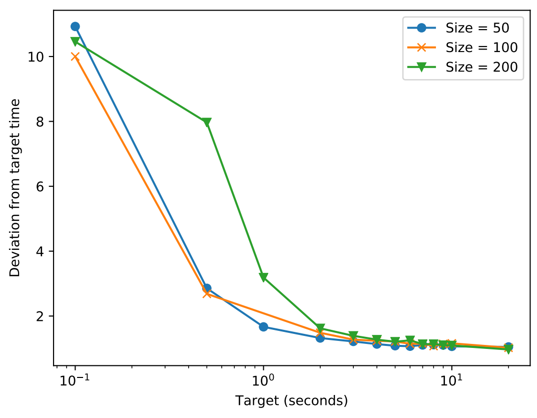
a World Topology



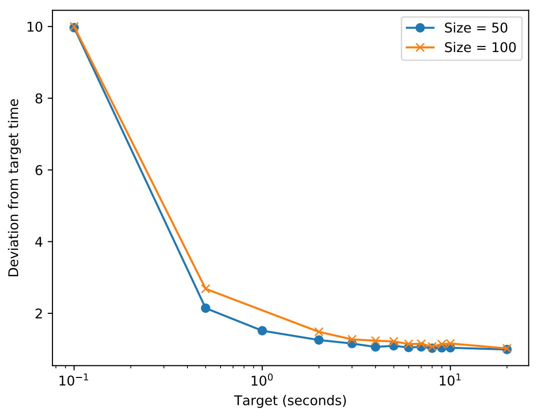
b Fully Connected Topology

Figure 5.5: Rate of blocks with multiple descendants.

case of target time 0.1 s, this means that the effective target time is about 1 s, while the latter configuration produces much less orphan blocks, and thus also less traffic on the network. This deviation decreases exponentially as target times approach 1 s or 2 s (depending on the network size), converging slowly to the target time as they keep going upwards. However, it is also interesting to note that a target time of 10 s or higher is required to keep the deviation close to 1.



a World Topology



b Fully Connected Topology

Figure 5.6: Deviation from the target time of the average block generation time.

5.9 Conclusions

In this chapter, we have proposed a methodology that uses container-based network emulation to study blockchain distributed systems that implement a consensus mechanism that is based on a proof of work. As a reference implementation, we have instrumented and modified an Ethereum client that allows to simulate the mining process realistically and to keep a network event log from which the blockchain and any fork events can be reconstructed.

We have validated the platform against a real Ethereum private testnet and defined metrics to analyze the system in terms of the generation of forks.

We have shown that it is possible to use our methodology to study this type of systems in different emulated network topologies up to hundreds of nodes, using just a small hardware testbed.

6

Conclusions and Future Work

The study of distributed systems proposes challenges that derive from the difficulty to build a realistic environment in which the effects of different network conditions and heterogeneous platforms can be modeled. This imposes the need for specific tools that help developers and researchers to properly understand the effects of the environment on the performance of the system. In this work, we have presented SherlockFog, a network emulation tool that focuses on the study of distributed applications in heterogeneous environments, using lightweight virtualization technologies that are implemented in the Linux kernel. Two main groups of applications have been studied: parallel applications for scientific computing using MPI and distributed systems that implement blockchain-based technologies such as cryptocurrencies.

The main contributions are the following:

- Latency modeling of a Fog/Edge Computing scenario to assess the performance degradation in several representative MPI applications for CFD: We have validated the methodology to show the accuracy of the latency modeling, while also showing that the performance degradation in the proposed scenario is sublinear with respect to the latency of the network.

- CPU performance modeling in IoT environments, also used in the context of executing MPI applications: We have used the IoT-LAB platform to validate a CPU emulation module for SherlockFog that extends the platform to properly model IoT platforms with heterogeneous computing power. We have also shown that since the CPU performance of this type of platforms is lower than traditional HPC, it is more resilient to changes in latency than the latter. This enables a potential use of IoT for scientific computing that could achieve a better use of already deployed platforms.

- Modeling of a popular PoW-based cryptocurrency to study characteristics of the blockchain in different network conditions: We have used SherlockFog as a platform to deploy dynamically-generated configurations to a set of nodes and execute a test network of the Ethereum cryptocurrency on an arbitrary network topology. We have modeled PoW using statistical properties of the game in order to model miners without the expensive computing involved. Scenarios of varying latencies show some limits of the cryptocurrency to the ability to process new blocks.

SherlockFog implements a custom language, *fog*, that allows the user to define repeatable experiments. It runs on commodity hardware, requiring no special interconnection technology. The tool can connect namespaces in one or more physical nodes, provided that the traffic is in the same physical network. The system can easily accommodate to additional nodes without user effort in order to scale the size of the emulated network. Unlike other similar tools, it makes use of standard operating system facilities that ease maintainability. Additionally, it allows CPU resources to be preallocated to isolate code execution, making it a good fit to study computationally intensive applications. Application code can be executed unmodified, acting as a development aid for many types of distributed systems.

Feature-wise, it allows the user to change the latency, bandwidth and packet loss of a link, which is important for Fog/Edge Computing environments. It also allows timers to be set to change network parameters at a specific point in time in a reproducible way. Using the actual software implementation allows the user to experiment with different runtime libraries and technologies without requiring code changes.

The tool supports several static routing strategies depending on the network topology that is to be emulated. This allows any arbitrary network scenario to be modelled.

The *fog* language also includes related tools to help the user create their network scenarios. A graph generator allows networks to be converted from popular exchange formats to the *fog* language or export them for visualization. It is also possible to generate several well-known topologies parametrically.

We have shown that SherlockFog is a platform that enhances the study of parallel and distributed systems on different network conditions, providing a controlled environment for testing, proposing experiments and implementing changes. This tool will prove to be a development aid for more scalable and resilient systems that take advantage of existing and new hardware deployments.

6.1 Future Work

With respect to the study of parallel applications, several open questions derive from this work:

- Evaluate churn in MPI applications, introducing necessary changes to library implementations.

  Currently, FT-MPI supports process recovery in MPI-1.2. No implementation of a newer version of MPI exists. Moreover, it is not possible to change the total number of nodes during a run.

- Introduce resiliency to packet loss into MPI library implementations and evaluate it.

  Most existing implementations bring computation to a halt if a packet is lost. Introducing a different transmission protocol could prove useful for IoT scenarios in which connectivity is more fail-prone.

- Study dynamic network conditions, such as run-time changes to latency, bandwidth and packet loss.

  The NetEm implementation supports the emulation of different network conditions that could be introduced in SherlockFog scenarios to model wireless and other types of links more accurately.

- Extend the study to other technologies for parallel computing.

  As our methodology is not closely tied to MPI libraries or implementations, it would be possible to use it to evaluate other technologies as well.

The application of SherlockFog to study blockchain-based systems also opens up many questions:

- Evaluate the scalability limits of the systems to properly model a complete cryptocurrency network.

  This requires a description of the complete physical and logical topologies, which involves using topology reconstruction techniques to indirectly obtain this information.

- Reproduce the work for other PoW-based cryptocurrencies.

  Our methodology uses characteristics of PoW-based systems but it is not tied to Ethereum. It would be an interesting analysis to compare the performance of other PoW-based cryptosystems in order to properly understand the implications of each design decision.

- Extend the platform to study Proof of Stake or other types of cryptosystems.

- Study the peer selection policy in Ethereum—used to build the logical topology of the network—to assess the effects on information propagation, in order to be able to propose novel strategies that improve the scalability of the network and reduce network concentration.
- Model attacks to the network such as cartel mining and selfish mining.

6.2 Publications Based on This Thesis

The following presentations and publications have been based on work that has been completed during this thesis:

- Geier M., Tessone C. J., Vanotti M, Vileriño S., González Márquez D., Mocskos E. (2019) **Using Network Emulation to study Blockchain Distributed Systems: The Ethereum Case**. Parallel Distributed Computing Conference 2019. (*in press*).
- Geier M., González Márquez D. and Mocskos E. (2018) **SherlockFog: a New Tool to Support Application Analysis in Fog and Edge Computing**. Cluster Computing. (*under review*).
- Geier M., Mocskos E. (2017) **SherlockFog: Finding Opportunities for MPI Applications in Fog and Edge Computing**. In: Mocskos E., Nesmachnow S. (eds) High Performance Computing. CARLA 2017. Communications in Computer and Information Science, vol 796. Springer, Cham.
- Da Silva M., Nesmachnow S., Geier M., Mocskos E., Angiolini J., Levi V., Cristobal A. (2014) **Efficient Fluorescence Microscopy Analysis over a Volunteer Grid/Cloud Infrastructure**. In: Hernández G. et al. (eds) High Performance Computing. CARLA 2014. Communications in Computer and Information Science, vol 485. Springer, Berlin, Heidelberg.
- Geier M. and Mocskos E. (2013) **Improving capabilities of P2P Volunteer Computing Platform**. Workshop on Dynamic Networks 2013.

List of Acronyms

cgroup	control group
AR	Augmented Reality
ARP	Address Resolution Protocol
API	Application Programming Interface
BoT	Bag-of-Tasks
BTC	Bitcoin
CFD	Computer Fluid Dynamics
CFS	Completely Fair Scheduler
CIDR	Classless Inter-Domain Routing
DCE	Direct Code Execution
DES	Discrete-Event Simulation
DFS	Depth-first Search
DNS	Domain Name System
FFT	Fast Fourier Transform
GHOST	Greedy Heaviest Observed Subtree
HPC	High Performance Computing
IaaS	Infrastructure as a Service
IoT	Internet of Things
IPv4	IP version 4
IPv6	IP version 6
JIT	Just-in-time
LAN	Local-area Network
MPI	Message Passing Interface
NAS	Numerical Aerospace Simulation

NTP	Network Time Protocol
NUMA	Non-Uniform Memory Architecture
ON	Open Node
PaaS	Platform as a Service
PDE	Partial Differential Equation
PI	Principal Investigator
PKI	Public Key Infrastructure
P2P	Point-to-point
PoW	Proof of Work
RSH	Remote Shell
SaaS	Software as a Service
SBC	Single-Board Computer
SDN	Software-Defined Network
SNMP	Simple Network Management Protocol
SSH	Secure Shell
TCP	Transport Control Protocol
UMA	Uniform Memory Architecture
UTXO	Unspent Transaction Output
VLAN	Virtual Local Area Network
VPN	Virtual Private Network
WAN	Wide-area Network

Bibliography

[1] L. Nussbaum, Contributions to experimentation on large scale distributed systems, Theses, Université Joseph-Fourier - Grenoble I (Dec. 2008).
URL https://tel.archives-ouvertes.fr/tel-00365394

[2] J. L. Welch, H. Attiya, Distributed Computing: Fundamentals, Simulations and Advanced Topics, McGraw-Hill, Inc., New York, NY, USA, 1998.

[3] J. Gustedt, E. Jeannot, M. Quinson, Experimental Methodologies for Large-Scale Systems: a Survey, Parallel Processing Letters 19 (3) (2009) 399–418. doi:10.1142/S0129626409000304.
URL http://hal.inria.fr/inria-00364180

[4] M. Geier, L. Nussbaum, M. Quinson, On the convergence of experimental methodologies for distributed systems: Where do we stand, in: WATERS - 4th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems, Jul 2013, Paris, France, 2013.

[5] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, ACM, New York, NY, USA, 2012, pp. 13–16. doi:10.1145/2342509.2342513.

[6] N. R. Herbst, S. Kounev, R. Reussner, Elasticity in cloud computing: What it is, and what it is not, in: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), USENIX, San Jose, CA, 2013, pp. 23–27.
URL https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet of Things Journal 3 (5) (2016) 637–646. doi:10.1109/JIOT.2016.2579198.

[8] M. Hirsch, C. Mateos, A. Zunino, Augmenting computing capabilities at the edge by jointly exploiting mobile devices: A survey, Future Generation Computer Systemsdoi:10.1016/j.future.2018.06.005.
URL http://dx.doi.org/10.1016/j.future.2018.06.005

[9] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd Edition, MIT Press, 1999.

[10] W. Gropp, E. Lusk, R. Thakur, Using MPI-2: Advanced Features of the Message-Passing Interface, 2nd Edition, MIT Press, 1999.

[11] P. S. Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann Publishers Inc., 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 2011.

[12] G. E. Fagg, J. J. Dongarra, Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world, in: J. Dongarra, P. Kacsuk, N. Podhorszki (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 346–353.

[13] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, The nas parallel benchmarks 5 (3) (1991) 63–73. `doi:10.1177/109434209100500306`.

[14] H. Jin, R. F. V. der Wijngaart, Performance characteristics of the multi-zone nas parallel benchmarks, Journal of Parallel and Distributed Computing 66 (5) (2006) 674 – 685, iPDPS '04 Special Issue. `doi:https://doi.org/10.1016/j.jpdc.2005.06.016`.
URL `http://www.sciencedirect.com/science/article/pii/S0743731505001644`

[15] A. Waheed, J. Yan, Parallelization of nas benchmarks for shared memory multiprocessors, Future Generation Computer Systems 15 (3) (1999) 353 – 363. `doi:10.1016/S0167-739X(98)00080-6`.
URL `http://www.sciencedirect.com/science/article/pii/S0167739X98000806`

[16] H. A. Hassan, S. A. Mohamed, W. M. Sheta, Scalability and communication performance of HPC on Azure Cloud, Egyptian Informatics Journal 17 (2) (2016) 175 – 182. `doi:10.1016/j.eij.2015.11.001`.
URL `http://www.sciencedirect.com/science/article/pii/S1110866515000523`

[17] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, B. Chapman, High performance computing using mpi and openmp on multi-core parallel systems, Parallel Computing 37 (9) (2011) 562 – 575, emerging Programming Paradigms for Large-Scale Scientific Computing. `doi:https://doi.org/10.1016/j.parco.2011.02.002`.
URL `http://www.sciencedirect.com/science/article/pii/S0167819111000159`

[18] W. Zhang, A. M. K. Cheng, J. Subhlok, Dwarfcode: A performance prediction tool for parallel applications, IEEE Transactions on Computers 65 (2) (2016) 495–507. `doi:10.1109/TC.2015.2417526`.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, SIGPLAN Not. 40 (6) (2005) 190–200. `doi:10.1145/1064978.1065034`.
URL `http://doi.acm.org/10.1145/1064978.1065034`

[20] A.-L. Barabási, R. Albert, Emergence of Scaling in Random Networks, Science 286 (1999) 509–512. `doi:10.1126/science.286.5439.509`.

[21] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, The NAS Parallel Benchmarks, Report RNR-94-007, Department of Mathematics and Computer Science, Emory University (Mar. 1994).

[22] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of things (iot): A vision, architectural elements, and future directions, Future Generation Computer Systems 29 (7) (2013) 1645 – 1660, including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications - Big Data, Scalable Analytics, and Beyond. `doi:10.1016/j.future.2013.01.010`.
URL `http://www.sciencedirect.com/science/article/pii/S0167739X13000241`

[23] M. N. Durrani, J. A. Shamsi, Volunteer computing: requirements, challenges, and solutions, Journal of Network and Computer Applications 39 (2014) 369 – 380. `doi:10.1016/j.jnca.2013.07.006`.
URL `http://www.sciencedirect.com/science/article/pii/S1084804513001665`

[24] E. Cesario, C. Mastroianni, D. Talia, Distributed volunteer computing for solving ensemble learning problems, Future Generation Computer Systems 54 (2016) 68–78. doi:10.1016/j.future.2015.07.010.
URL http://www.sciencedirect.com/science/article/pii/S0167739X15002332

[25] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, T. Watteyne, Fit iot-lab: A large scale open experimental iot testbed, in: 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), 2015, pp. 459–464. doi:10.1109/WF-IoT.2015.7389098.

[26] R. Hinden, B. Carpenter, L. Masinter, Format for literal ipv6 addresses in url's, RFC 2732, RFC Editor (December 1999).
URL http://www.rfc-editor.org/rfc/rfc2732.txt

[27] R. Hinden, S. Deering, Ip version 6 addressing architecture, RFC 4291, RFC Editor (February 2006).
URL http://www.rfc-editor.org/rfc/rfc4291.txt

[28] Global ethereum nodes distribution, http://ethernodes.org/network/1, accessed: February 4, 2019.

[29] A. D. Kshemkalyani, M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, 1st Edition, Cambridge University Press, New York, NY, USA, 2008.

[30] M. Singhal, N. G. Shivaratri, Advanced Concepts in Operating Systems, McGraw-Hill, Inc., New York, NY, USA, 1994.

[31] A. S. Tanenbaum, M. v. Steen, Distributed Systems: Principles and Paradigms (2Nd Edition), Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[32] A. M. Goscinski, Distributed operating systems - the logical design, 1991.

[33] P. V. Mockapetris, Domain names - implementation and specification (1987).

[34] O. Beaumont, L. Bobelin, H. Casanova, P.-N. Clauss, B. Donassolo, L. Eyraud-Dubois, S. Genaud, S. Hunold, A. Legrand, M. Quinson, C. Rosa, L. Schnorr, M. Stillwell, F. Suter, C. Thiery, P. Velho, J.-M. Vincent, J. Won, Young, Towards Scalable, Accurate, and Usable Simulations of Distributed Applications and Systems, Rapport de recherche RR-7761, INRIA (Oct. 2011).
URL http://hal.inria.fr/inria-00631141

[35] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, M. Wawrzoniak, Operating system support for planetary-scale network services, in: Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1, NSDI'04, USENIX Association, Berkeley, CA, USA, 2004, pp. 19–19.
URL http://dl.acm.org/citation.cfm?id=1251175.1251194

[36] Planet-lab world view, https://www.planet-lab.org/generated/World50.png, last checked: February 4, 2019.

[37] D. R. Choffnes, F. E. Bustamante, Taming the torrent: A practical approach to reducing cross-isp traffic in peer-to-peer systems, SIGCOMM Comput. Commun. Rev. 38 (4) (2008) 363–374. doi:10.1145/1402946.1403000.
URL http://doi.acm.org/10.1145/1402946.1403000

[38] J. Banks, J. Carson, B. L. Nelson, D. Nicol, Discrete-Event System Simulation (4th Edition), 4th Edition, Prentice Hall, 2004.
URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0131446797

[39] ns-3 Overview, https://www.nsnam.org/docs/ns-3-overview.pdf, last checked: February 4, 2019.

[40] ns-3 Direct Code Execution, https://www.nsnam.org/overview/projects/direct-code-execution/, last checked: February 4, 2019.

[41] H. Casanova, A. Giersch, A. Legrand, M. Quinson, F. Suter, Versatile, scalable, and accurate simulation of distributed applications and platforms, Journal of Parallel and Distributed Computing 74 (10) (2014) 2899–2917.
URL http://hal.inria.fr/hal-01017319

[42] A. Degomme, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, F. Suter, Simulating mpi applications: the smpi approach, IEEE Transactions on Parallel and Distributed Systems PP (99) (2017) 1–1. doi:10.1109/TPDS.2017.2669305.

[43] Dimemas, http://tools.bsc.es/dimemas, last checked: February 4, 2019.

[44] S. Hemminger, Network emulation with NetEm, in: M. Pool (Ed.), LCA 2005, Australia's 6th national Linux conference (linux.conf.au), Linux Australia, Linux Australia, Sydney NSW, Australia, 2005.
URL http://developer.osdl.org/shemminger/netem/LCA2005_paper.pdf

[45] D. Pediaditakis, C. Rotsos, A. W. Moore, Faithful reproduction of network experiments, in: Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '14, ACM, New York, NY, USA, 2014, pp. 41–52. doi:10.1145/2658260.2658274.

[46] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, D. Becker, Scalability and accuracy in a large-scale network emulator, SIGOPS Oper. Syst. Rev. 36 (SI) (2002) 271–284. doi:10.1145/844128.844154.

[47] B. Lantz, B. Heller, N. McKeown, A network in a laptop: Rapid prototyping for software-defined networks, in: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets-IX, ACM, New York, NY, USA, 2010, pp. 19:1–19:6. doi:10.1145/1868447.1868466.

[48] P. Wette, M. Dräxler, A. Schwabe, Maxinet: Distributed emulation of software-defined networks, in: Networking Conference, 2014 IFIP, 2014, pp. 1–9. doi:10.1109/IFIPNetworking.2014.6857078.

[49] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1) (1998) 359–392. doi:10.1137/S1064827595287997.

[50] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar, An integrated experimental environment for distributed systems and networks, in: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, USENIX Association, Boston, MA, 2002, pp. 255–270.

[51] K. Ashton, That 'Internet of Things' Thing, RFID Journal.

[52] H. Sundmaeker, P. Guillemin, P. Friess, S. Woelfflé (Eds.), Vision and Challenges for Realising the Internet of Things, Publications Office of the European Union, Luxembourg, 2010. doi:10.2759/26127.

[53] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of cloud computing, Commun. ACM 53 (4) (2010) 50–58. doi:10.1145/1721654.1721672.
URL http://doi.acm.org/10.1145/1721654.1721672

[54] D. Huang, H. Wu, Chapter 3 - mobile cloud service models, in: D. Huang, H. Wu (Eds.), Mobile Cloud Computing, Morgan Kaufmann, 2018, pp. 65 – 85. doi:https://doi.org/10.1016/B978-0-12-809641-3.00004-1.
URL http://www.sciencedirect.com/science/article/pii/B9780128096413000041

[55] Message Passing Interface Forum, Mpi: A message-passing interface standard, version 3.1, Specification (June 2015).
URL http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[56] A. Brown, G. Wilson, The Architecture Of Open Source Applications, lulu.com, 2011.
URL http://www.aosabook.org/en/

[57] G. E. Fagg, J. Pjesivac-grbovic, G. Bosilca, J. J. Dongarra, E. Jeannot, Flexible collective communication tuning architecture applied to open mpi, in: In 2006 Euro PVM/MPI, 2006.

[58] J. S. Vetter, M. O. McCracken, Statistical scalability analysis of communication operations in distributed applications, SIGPLAN Not. 36 (7) (2001) 123–132. doi:10.1145/568014.379590.
URL http://doi.acm.org/10.1145/568014.379590

[59] M. Geier, E. Mocskos, Sherlockfog: Finding opportunities for mpi applications in fog and edge computing, in: E. Mocskos, S. Nesmachnow (Eds.), High Performance Computing, Springer International Publishing, Cham, 2017, pp. 185–199.

[60] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, N. McKeown, Reproducible network experiments using container-based emulation, in: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, CoNEXT '12, ACM, New York, NY, USA, 2012, pp. 253–264. doi:10.1145/2413176.2413206.

[61] P. Turner, B. B. Rao, N. Rao, Cpu bandwidth control for cfs, in: Proceedings of the Linux Symposium, 2010, pp. 245–254.
URL http://www.linuxsymposium.org/LS_2010_Proceedings_Draft.pdf

[62] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual, Intel Corporation, 2018.

[63] B. Brandfass, T. Alrutz, T. Gerhold, Rank reordering for mpi communication optimization, Computers & fluids 80 (2013) 372 – 380, selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011. doi:10.1016/j.compfluid.2012.01.019.
URL http://www.sciencedirect.com/science/article/pii/S004579301200028X

[64] K. Dichev, V. Rychkov, A. Lastovetsky, Two algorithms of irregular scatter/gather operations for heterogeneous platforms, in: Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface, EuroMPI'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 289–293.
URL http://dl.acm.org/citation.cfm?id=1894122.1894162

[65] G. Mercier, J. Clet-Ortega, Towards an efficient process placement policy for mpi applications in multicore environments, in: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 104–115. doi:10.1007/978-3-642-03770-2_17.

[66] J. Navaridas, J. A. Pascual, J. Miguel-Alonso, Effects of job and task placement on parallel scientific applications performance, in: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009, pp. 55–61. doi:10.1109/PDP.2009.53.

[67] D. Mills, J. Martin, J. Burbank, W. Kasch, Network time protocol version 4: Protocol and algorithms specification, RFC 5905, RFC Editor (June 2010).
URL http://www.rfc-editor.org/rfc/rfc5905.txt

[68] T. M. J. Fruchterman, E. M. Reingold, Graph drawing by force-directed placement, Softw. Pract. Exper. 21 (11) (1991) 1129–1164. doi:10.1002/spe.4380211102.

[69] J. J. Dongarra, P. Luszczek, A. Petitet, The linpack benchmark: past, present and future, Concurrency and Computation: Practice and Experience 15 (9) 803–820. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728, doi:10.1002/cpe.728.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728

[70] S. J. Johnston, P. J. Basford, C. S. Perkins, H. Herry, F. P. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, S. J. Cox, J. Singer, Commodity single board computer clusters and their applications, Future Generation Computer Systems 89 (2018) 201 – 212. doi:10.1016/j.future.2018.06.048.
URL http://www.sciencedirect.com/science/article/pii/S0167739X18301833

[71] S. J. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, N. S. O'Brien, Iridis-pi: a low-cost, compact demonstration cluster, Cluster Computing 17 (2) (2014) 349–358. doi:10.1007/s10586-013-0282-7.

[72] R. Mayer, L. Graser, H. Gupta, E. Saurez, U. Ramachandran, Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures, in: IEEE Fog World Congress, FWC 2017, Santa Clara, CA, USA, October 30 - Nov. 1, 2017, IEEE, 2017, pp. 1–6. doi:10.1109/FWC.2017.8368525.
URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8364427

[73] T. Buchert, L. Nussbaum, J. Gustedt, Methods for emulation of multi-core cpu performance, in: 2011 IEEE International Conference on High Performance Computing and Communications, 2011, pp. 288–295. doi:10.1109/HPCC.2011.45.

[74] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008).

[75] F. Tschorsch, B. Scheuermann, Bitcoin and beyond: A technical survey on decentralized digital currencies, IEEE Communications Surveys Tutorials 18 (3) (2016) 2084–2123. doi:10.1109/COMST.2016.2535718.

[76] W. C. Wei, Liquidity and market efficiency in cryptocurrencies, Economics Letters 168 (2018) 21 – 24. doi:10.1016/j.econlet.2018.04.003.
URL http://www.sciencedirect.com/science/article/pii/S0165176518301320

[77] P. Tasca, C. J. Tessone, Taxonomy of blockchain technologies. principles of identification and classification, arXiv:1708.04872.

[78] Z. Zheng, S. Xie, H. Dai, X. Chen, H. Wang, An overview of blockchain technology: Architecture, consensus, and future trends, in: 2017 IEEE International Congress on Big Data (BigData Congress), 2017, pp. 557–564. `doi:10.1109/BigDataCongress.2017.85`.

[79] A. Reyna, C. Martín, J. Chen, E. Soler, M. Díaz, On blockchain and its integration with iot. challenges and opportunities, Future Generation Computer Systems 88 (2018) 173 – 190. `doi:10.1016/j.future.2018.05.046`.
URL `http://www.sciencedirect.com/science/article/pii/S0167739X17329205`

[80] L. Zhou, L. Wang, Y. Sun, Mistore: a blockchain-based medical insurance storage system, Journal of Medical Systems 42 (8) (2018) 149. `doi:10.1007/s10916-018-0996-4`.

[81] H. Li, L. Zhu, M. Shen, F. Gao, X. Tao, S. Liu, Blockchain-based data preservation system for medical data, Journal of Medical Systems 42 (8) (2018) 141. `doi:10.1007/s10916-018-0997-3`.

[82] M. P. Caro, M. S. Ali, M. Vecchio, R. Giaffreda, Blockchain-based traceability in agri-food supply chain management: A practical implementation, in: 2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany), 2018, pp. 1–4. `doi:10.1109/IOT-TUSCANY.2018.8373021`.

[83] F. Lamberti, V. Gatteschi, C. Demartini, M. Pelissier, A. Gomez, V. Santamaria, Blockchains can work for car insurance: Using smart contracts and sensors to provide on-demand coverage, IEEE Consumer Electronics Magazine 7 (4) (2018) 72–81. `doi:10.1109/MCE.2018.2816247`.

[84] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün, On scaling decentralized blockchains, in: Proc. 3rd Workshop on Bitcoin and Blockchain Research, 2016.

[85] G. Wood, Ethereum yellow paper (2014).

[86] V. Buterin, A next-generation smart contract and decentralized application platform, white paper.

[87] Y. Sompolinsky, A. Zohar, Secure high-rate transaction processing in bitcoin, in: R. Böhme, T. Okamoto (Eds.), Financial Cryptography and Data Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 507–527.

[88] I. Eyal, E. G. Sirer, Majority is not enough: Bitcoin mining is vulnerable, in: International Conference on Financial Cryptography and Data Security, Springer, 2014, pp. 436–454.

[89] R. Jansen, N. Hopper, Shadow: Running tor in a box for accurate and efficient experimentation, in: Proceedings of the 19th Symposium on Network and Distributed System Security (NDSS), Internet Society, 2012.

[90] A. Miller, R. Jansen, Shadow-bitcoin: Scalable simulation via direct execution of multi-threaded applications, in: Proceedings of the 8th USENIX Conference on Cyber Security Experimentation and Test, CSET'15, USENIX Association, Berkeley, CA, USA, 2015, pp. 7–7.
URL `http://dl.acm.org/citation.cfm?id=2831120.2831127`

[91] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, S. Capkun, On the security and performance of proof of work blockchains, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, ACM, New York, NY, USA, 2016, pp. 3–16. `doi:10.1145/2976749.2978341`.
URL `http://doi.acm.org/10.1145/2976749.2978341`

[92] C. Decker, R. Wattenhofer, Information propagation in the bitcoin network, in: IEEE P2P 2013 Proceedings, IEEE, 2013, pp. 1–10.

[93] M. Vanotti, Un avance hacia entornos de gran escala para experimentos con criptomonedas (2016).

[94] S. Vileriño, Estudio de los límites de generación de bloques en blockchain (2017).

A

SherlockFog Reference

A.1 Software Requirements

SherlockFog is implemented in Python version 3 and has dependencies on some Python modules and shell commands. The following Python modules are required:

- `networkx`: graph handling.
- `matplotlib`: graph visualization and plotting, also a dependency for `networkx`.
- `paramiko`: SSH connection handling.

The following shell commands are called from the SherlockFog code:

- `ip`: handles virtual network interface creation, address discovery, routing tables, ARP tables, network namespaces.
- `tc`: traffic shaping handling.
- `cgcreate`: handles cgroup creation.
- `cgset`: handles CPU sets and other cgroup parameters.
- `cgdelete`: removes cgroups.
- `cgexec`: command execution inside a cgroup.
- `ssh`: used to execute commands on an interactive shell inside a virtual node.
- `sshd`: OpenSSH is instantiated on every virtual node to accept connections.
- `lscpu`: CPU topology discovery.
- `unshare`: UTS namespace creation.

A.2 Command Line Arguments

Calling SherlockFog with the `-h` or `--help` flag produces the following output:

```
usage: sherlockfog.py [-h] [--dry-run [DRY_RUN]]
                      [--real-host-list [REAL_HOST_LIST]]
                      [-D DEFINE [DEFINE ...]] [--base-prefix [BASE_PREFIX]]
                      [--base-adm-prefix [BASE_ADM_PREFIX]]
                      [--use-iface-prefix [USE_IFACE_PREFIX]]
                      [--node-name-prefix [NODE_NAME_PREFIX]]
                      [--use-adm-ns [USE_ADM_NS]]
                      [--routing-algo [{shortest_path,tree_subnets,world_topo}]]
                      [--adm-iface-addr [ADM_IFACE_ADDR]]
                      [--cpu-exclusive [CPU_EXCLUSIVE]]
                      TOPO

Setup Random Topology on Commodity Hardware (SherlockFog)

positional arguments:
  TOPO                  Topology script

optional arguments:
  -h, --help            show this help message and exit
  --dry-run [DRY_RUN]   Dry-run (do not connect, build topology locally)
  --real-host-list [REAL_HOST_LIST]
                        Pool of IPs to assign nodes to (use {nextRealHost})
  -D DEFINE [DEFINE ...], --define DEFINE [DEFINE ...]
                        Define key=value in execution context
  --base-prefix [BASE_PREFIX]
                        Base network prefix for namespace IPs (CIDR notation)
  --base-adm-prefix [BASE_ADM_PREFIX]
                        Base prefix for administrative network (CIDR notation)
  --use-iface-prefix [USE_IFACE_PREFIX]
                        Use node prefix for virtual interface names (default:
                        False)
  --node-name-prefix [NODE_NAME_PREFIX]
                        Define node name prefix (default: n{num})
  --use-adm-ns [USE_ADM_NS]
                        Setup administrative private network
  --routing-algo [{shortest_path,tree_subnets,world_topo}]
                        Set routing algorithm (default: shortest_path)
  --adm-iface-addr [ADM_IFACE_ADDR]
                        Outgoing address for administrative network (default:
                        IP of default route's interface)
  --cpu-exclusive [CPU_EXCLUSIVE]
                        Setup exclusive access to a single CPU core for each
                        virtual host (default: True)
```

Considerations, caveats and bugs:

- `--real-host-list`: this argument is optional in combination with either `--dry-run` or a topology script that doesn't create virtual nodes. The magic variable `{nextRealHost}` is defined in the main execution context as the next host in that list. It also accepts `-`, which denotes *standard input*. Defining less hosts than the number of requested virtual nodes will result in an error.

- `--cpu-exclusive`: SherlockFog reads the topology of the host using the `lscpu` command. It processes NUMA nodes and cores, and selects the core to be assigned to a particular virtual host by iterating the cores of each NUMA node in a round-robin fashion. If more virtual nodes are instantiated in a given host than the number of cores, setting the assigned core for exclusive access for the second time will result in an error.

- `--routing-algo`: the `world_topo` option relies on the names of the virtual nodes to properly define routing tables. Names **must** be defined following this template, where `{n}` is the node number and `{ccTLD}` is the top-level domain for a given country (in uppercase letters):

  - `h{n}`: end nodes.

    Must be connected to a single virtual node, which must be an intermediate switch (node degree 1). The prefix letter `h` may be changed using the `--node-name-prefix` argument.
  - `s{n}`: intermediate switches.

    Must be connected only to a world backbone switch and an end node (node degree 2).
  - `s{n}-{ccTLD}`: world backbone switch for country `{ccTLD}`.

    Must be connected to every other world backbone switch (node degree equals to the number of countries plus the number of intermediate switches for that country).

  Additionally, virtual nodes **must** define a `country` property to be able to properly identify them using the `set-node-property` instruction in code.

- Using `--routing-algo tree_subnets` on a graph with loops may result in SherlockFog hanging.

- Using `--adm-iface-addr` with an address that corresponds to a different network interface than the default route that is used to connect to the hosts will result in the administrative network not being able to connect to any virtual node.

- Macvlan doesn't work on non-Ethernet interfaces (e.g. InfiniBand or the loopback interface).

- SherlockFog **does not** support IPv6.

A.3 Scripting Language

The *fog* scripting language allows the user to define, configure and initialize a virtual topology by executing repeatable experiment scripts. It also allows client code to be executed on top of the virtual infrastructure. The only control structure is the `for` command, which can be nested. The execution environment (class `ExecutionEnvironment`) keeps track of the state variables of the current scope and the topology graph. There is no conditional execution, which has to be handled using external scripts.

A syntactically correct *fog* program conforms to the following EBNF grammar:

```
Program       = { line | for | comment | eol };

line          = spaces, command, eol
command       = for_cmd | def_cmd | let_cmd | connect_cmd | set_delay_cmd |
                set_bw_cmd | set_nprop_cmd | shell_cmd | shelladm_cmd |
                buildnet_cmd | savegraph_cmd | include_cmd |
                run_cmd, runas_cmd, runadm_cmd;
comment       = spaces, "#", text, eol;
for_decl      = "for", space, id, space, "in", range, space, "do";
for_cmd       = for_decl, space, command;
for           = for_decl, eol, Program, "end for";
range         = number, "..", number, [ "..", number ];

def_cmd       = "def", space, id, [ space, ip_addr ];
let_cmd       = "let", space, id, space, expr;
connect_cmd   = "connect", space, id, space, id, [ space, expr ],
                { space, kwarg };
set_delay_cmd = "set-delay", space, [ at ], ( link | all ), rate;
set_bw_cmd    = "set-bandwidth", space, [ at ], ( link | all ), rate;
set_nprop_cmd = "set-node-property", space, id, space, id, space, value;
savegraph_cmd = "save-graph", space, value;
include_cmd   = "include", space, value;
shell_cmd     = "shell", [ space, id ];
shelladm_cmd  = "shelladm";
buildnet_cmd  = "build-network";
run_cmd       = "run", space, id, space, value;
runas_cmd     = "runas", space, id, space, id, space, value;
runadm_cmd    = "runadm", space, value;

link          = id, space, id;
ip_addr       = { digit }, ".", { digit }, ".", { digit }, ".", { digit };
at            = "at=", number, space;
number        = digit_nonzero, { "0" | digit_nonzero };
digit_nonzero = "[1-9]";
kwarg         = id, "=", expr;
value         = ? all visible characters ? - " ";
id            = "[A-Za-z_]", { "[0-9A-Za-z_-]" };
expr          = "[0-9A-Za-z_]", { "[0-9A-Za-z_-]" };
all           = "all";
spaces        = "[ \t]+";
space         = " ";
eol           = "\n";
```

The language has block scopes, but each command is matched within a single line. The following commands are defined:

- `def vnode`: defines a new virtual node called `vnode`.

  This instruction comprises creating a network namespace with the same name in one of the hosts of the real host pool and adding it to the topology.

- `let var value`: defines a syntactic replacement for expression `{var}` to `value` in the current execution context.
- `for var in start..end..step do cmd_list`: executes `cmd_list` in a loop, binding `{var}` to each value specified by `start..end..step`.
- `include file`: reads and executes every line in `file` in the current execution context.
- `runas vnode user cmd`: executes `cmd` as user `user` in `vnode`.
- `run vnode cmd`: executes `cmd` as `root` in `vnode`.
- `set-delay vnode1 vnode2 delay`: sets link delay between `vnode1` and `vnode2` to `delay`.
- `set-bandwidth vnode1 vnode2 bandwidth`: sets link bandwidth between `vnode1` and `vnode2` to `bandwidth`.
- `connect vnode1 vnode2 delay`: connects `vnode1` to `vnode2`, optionally setting the new link's delay to `delay`.

  This instruction defines new virtual interfaces in `vnode1` and `vnode2`, assigning IP addresses from a newly unassigned P2P subnet to both endpoints.
- `build-network`: this instruction is very important as it defines the routing and ARP tables in every virtual node, using the previously defined topology. The algorithm used to generate these rules depends on the value of the `-routing-algo` option.

  Failing to execute this command results in the virtual network not being able to route traffic unless a set of rules is defined manually. It also defines the containers' `/etc/hosts` file to be able to resolve the names of every node in the network. Note that every container has the same version of this file, but it must be replicated to be bound to each network namespace, which is done automatically by this command.
- `save-graph filename`: saves the current topology to `filename`. This command uses NetworkX and supports every format that is supported by this module.
- `set-node-property vnode prop value`: defines in `vnode` a node property with name `prop` and value `value`.
- `shell vnode`: starts a shell in virtual node `vnode`, or in the coordinator if unspecified.
- `shelladm`: starts a shell in the administrative virtual node. This command has no effect if the administrative interface has not been initialized.
- `runadm cmd`: executes `cmd` as `root` in the administrative virtual node. This command has no effect if the administrative interface has not been initialized.

Additionally, a macro system is used to query context information, which includes topology objects, scoped variables or "magic" arguments. Language expressions that enclose identifiers between brackets are replaced by their corresponding value before evaluation. These identifiers can appear anywhere except as part of the definition of a `for` command. A few examples follow:

- `run n0 ./cmd {n}` $\rightarrow$ replaces `{n}` with scoped variable `n`

- `run n{i} ./cmd {n0.default_iface.ip}` → this command requires two substitutions. It is executed on virtual host `n{i}`, which is resolved by replacing `{i}` with its scoped value, then appending it to the string `n`. For example, if `{i}` were `adm`, the resulting name would be `nadm`. Finally, the string `{n0.default_iface.ip}` is resolved by taking `n0` as a node object and navigating its attributes (it is evaluated within the Python interpreter). In this case, it will output the IP address of the default (first) virtual interface of `n0`.
- "Magic" arguments:
  - `def n0 {nextRealHost}` → `{nextRealHost}` resolves to the next IP address in the real host list. This address is removed from the list. It is also the default value of the optional IP argument.
  - `run n0 ./cmd {hostList}` → the `{hostList}` string resolves to a sorted, space-separated list of virtual node names, including every node in the topology (even unreachable nodes).
  - `run n0 {pwd}/myscript.py` → `{pwd}` resolves to the current working directory of the coordinator (equivalent to running the `pwd` command locally). This is useful to find scripts in a directory tree which is shared to all physical nodes.