(12) UK Patent (19) GB (11) 2625227 (13) B

(45) Date of B Publication 19.02.2025

(54) Title of the Invention: Performing tasks in distributed computer systems

(51) INT CL: ***G06F 9/50*** (2006.01) ***H04L 9/00*** (2022.01)

(21) Application No: **2403572.7**

(22) Date of Filing: **12.03.2024**

(30) Priority Data:
(31) **2309516** (32) **23.06.2023** (33) **GB**

(43) Date of A Publication **12.06.2024**

(56) Documents Cited:
CN 117251882 A **US 20230060916 A1**
The Consensus Machine: Formalising Consensus in the Presence of Malign Agents, Roscoe et al, Theories of programming and Formal Methods, Lecture Notes in Computer Science, Lect. Notes Computer 2023-09-08, Springer Nature Switzerland Cham, ISSN 0302-9743

(58) Field of Search:
As for published application 2625227 A viz:
INT CL **G06F, G06Q, H04L**
Other: **SEARCH-PATENT**
updated as appropriate

Additional Fields
Other: **None**

(72) Inventor(s):
Andrew William Roscoe
Pedro Antonino

(73) Proprietor(s):
The Blockhouse Technology Limited
2 Staverton Road, Oxford, Oxfordshire, OX2 6XJ, United Kingdom

(74) Agent and/or Address for Service:
Dehns
10 Old Bailey, London, EC4M 7NG, United Kingdom

GB 2625227 B

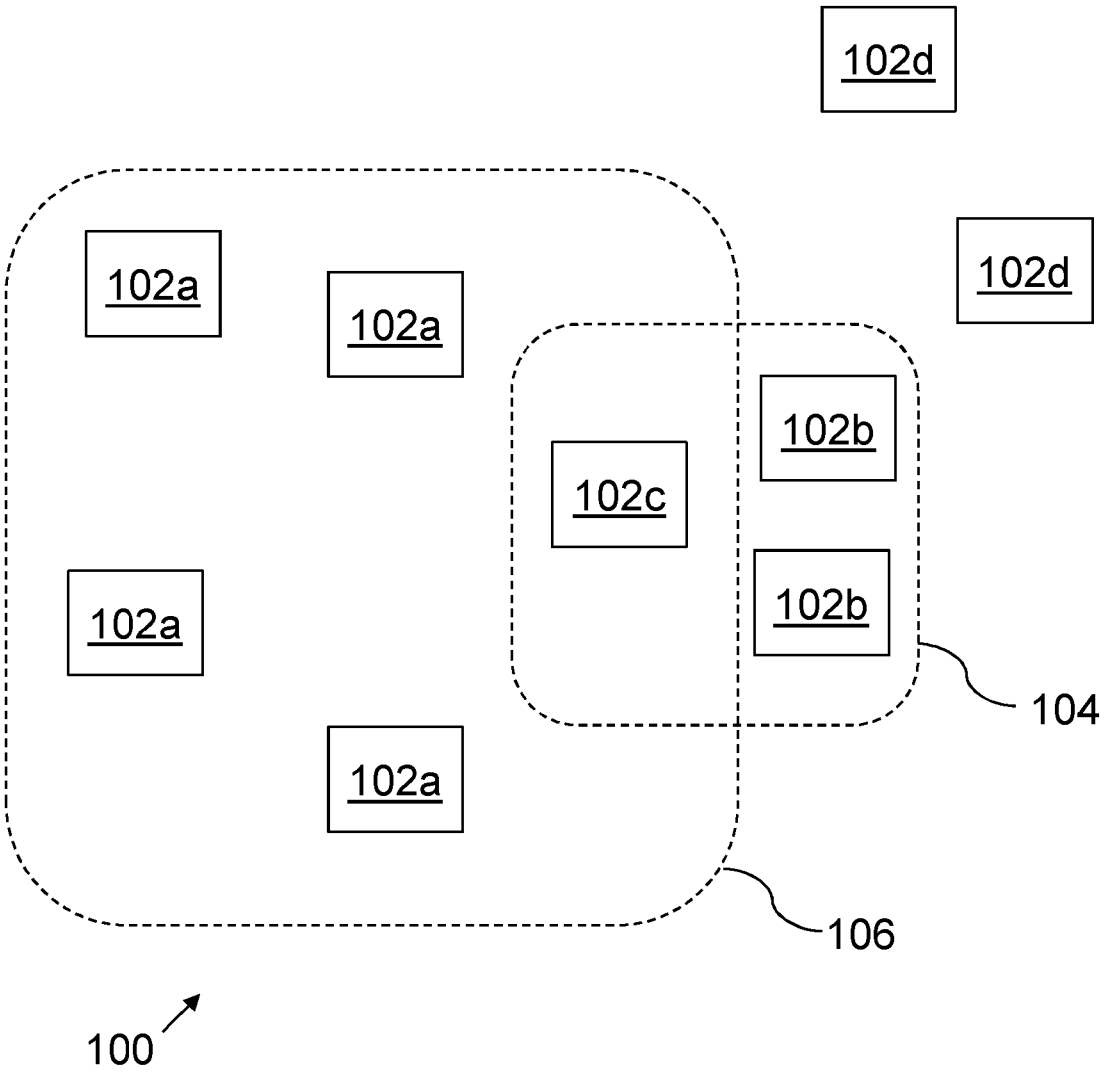


FIG. 1

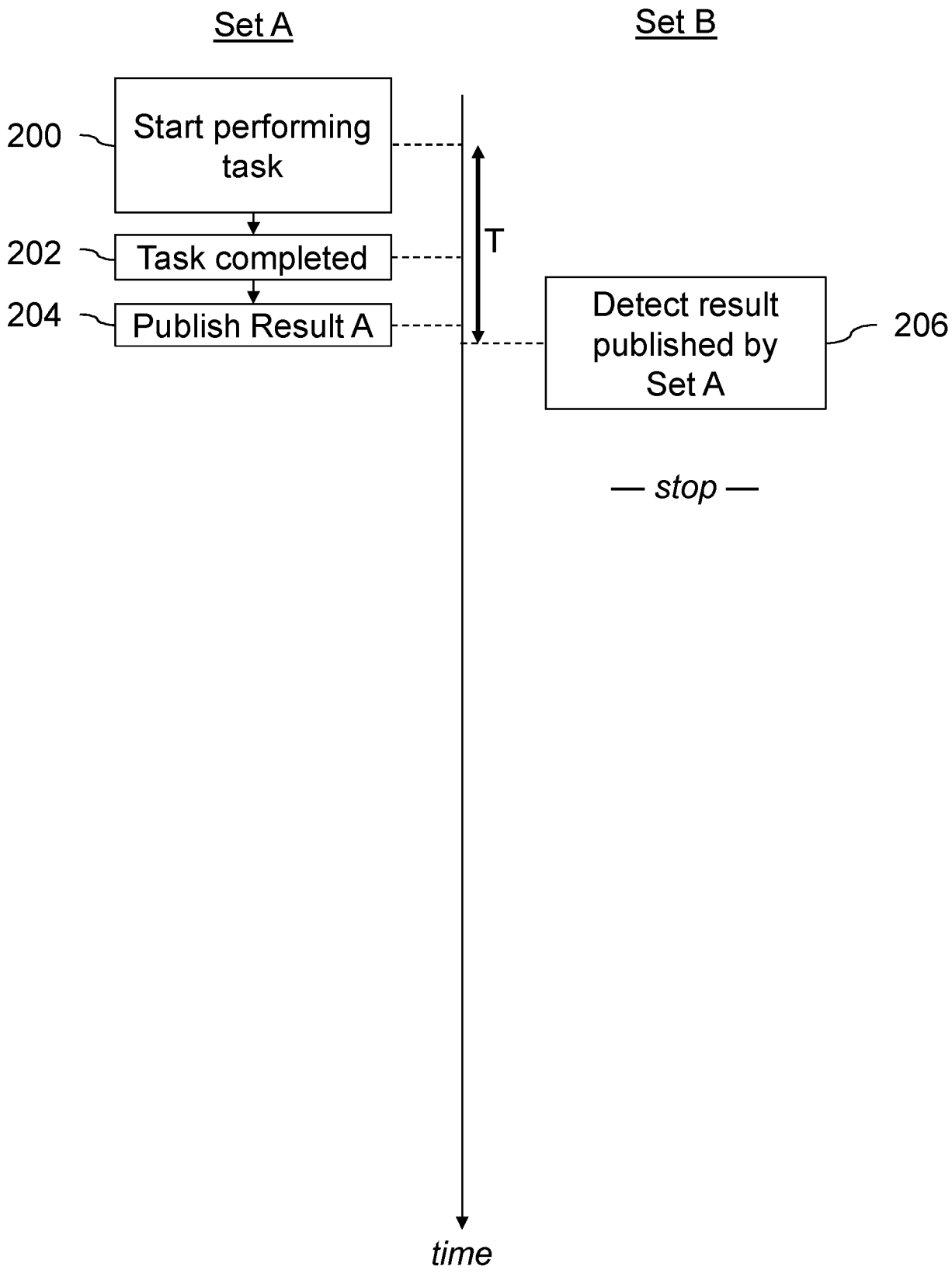

Set A
Set B
200
Start performing task
202
Task completed
204
Publish Result A
T
Detect result published by Set A
206
— stop —
time

FIG. 2

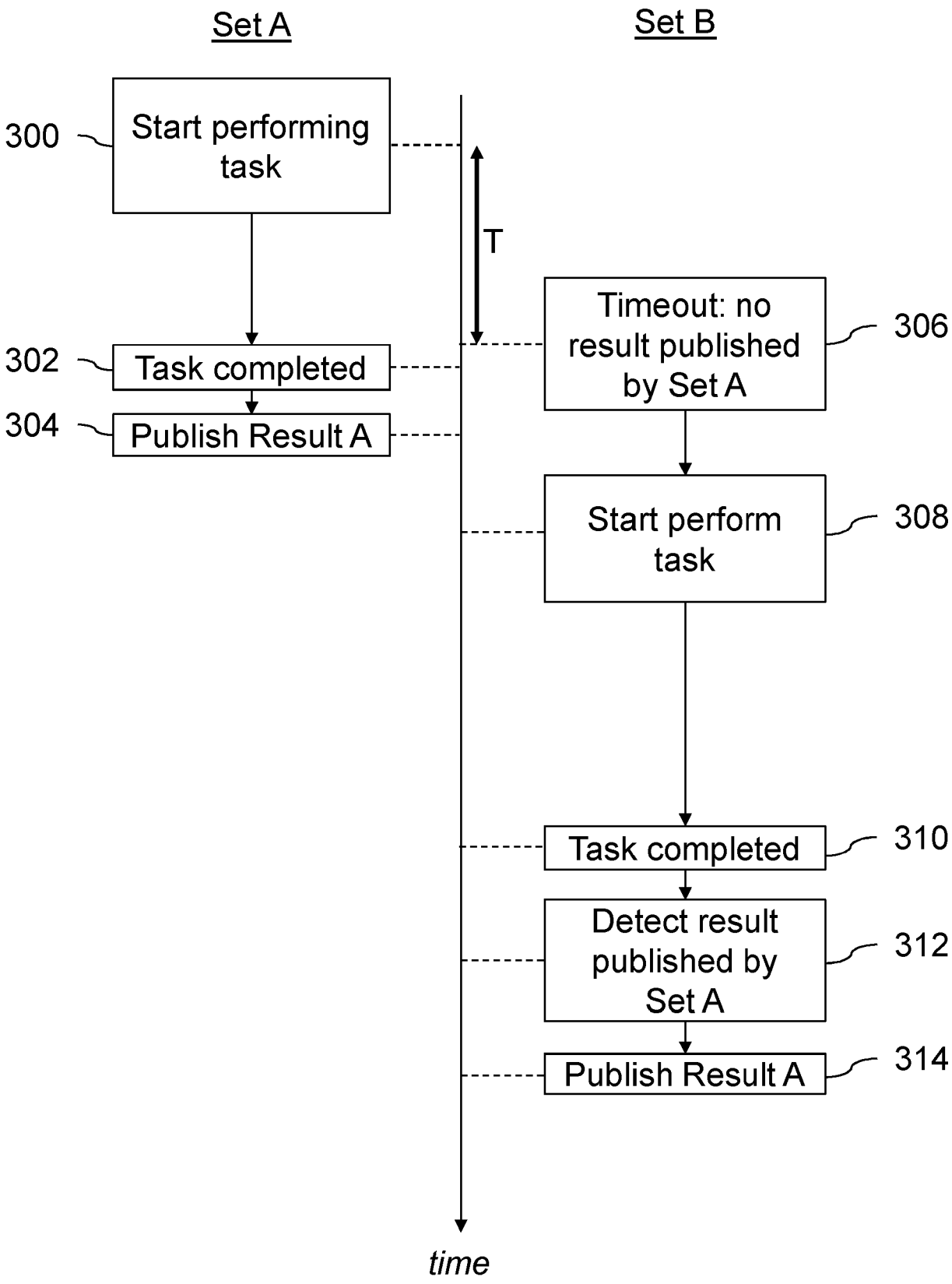

Set A
Set B
300
Start performing task
T
302
Task completed
304
Publish Result A
Timeout: no result published by Set A
306
Start perform task
308
Task completed
310
Detect result published by Set A
312
Publish Result A
314
time

FIG. 3

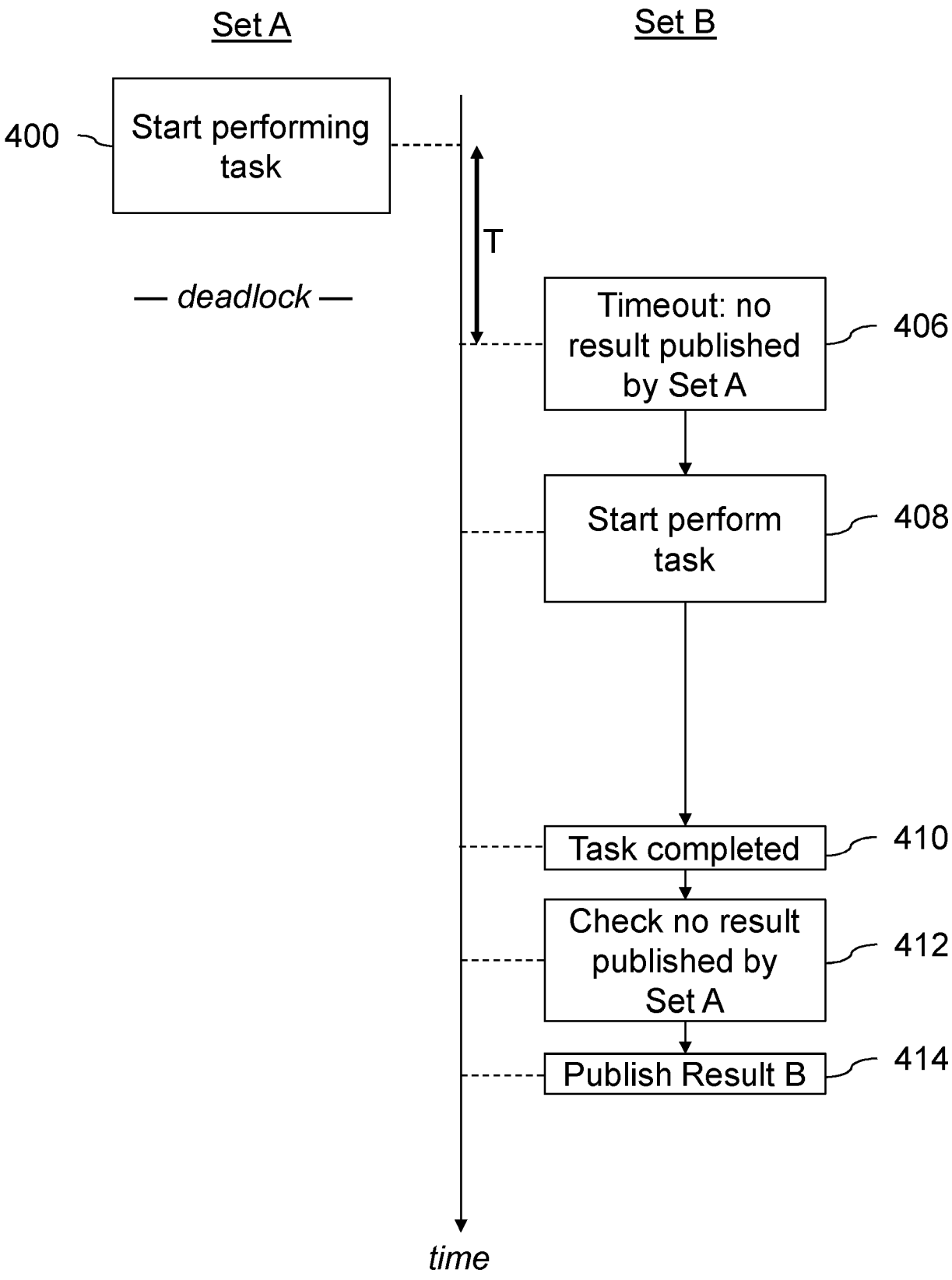

Set A
Set B
400
Start performing task
T
— deadlock —
Timeout: no result published by Set A
406
Start perform task
408
Task completed
410
Check no result published by Set A
412
Publish Result B
414
time

FIG. 4

Performing Tasks In Distributed Computer Systems

This invention relates to systems and methods for performing tasks in distributed computer systems.

BACKGROUND OF THE INVENTION

In a distributed computer system, such as a blockchain, computing devices of the system can interact concurrently by communicating over a network, such as the Internet, to perform tasks. In decentralized distributed systems, this occurs without the presence of a central controller or arbiter to manage the interactions between device and to define trust relationships between them.

In some distributed systems, a set of one or more computing devices may be delegated to perform a particular task for the system. The set of computing devices may be defined such that the set is guaranteed or nearly guaranteed never to output a "wrong" decision. However, a problem can arise if the set of devices is unable to complete the task in a timely manner. This may be especially problematic in a decentralized system, where there is no central server that might otherwise step in under such circumstances. For example, in a blockchain, a subset of the blockchain nodes may be delegated the task of deciding whether a candidate block should be added to the blockchain, and may be defined such that it is infeasible for the subset to dishonestly agree to add a malicious block; however, if these nodes cannot reach a timely decision at all, this may compromise the proper functioning of the blockchain.

Embodiments of the present invention seek to address this challenge.

SUMMARY OF THE INVENTION

From a first aspect, the invention provides a distributed computer system for performing a task, wherein the distributed computer system comprises a plurality of networked computing devices, and wherein:

a first set of the computing devices is configured to start performing a task, and to output a final result of the task after the task is completed;

a second set of the computing devices, different from the first set, is configured to determine whether the first set has output a final result of the task within a timeout period, and, if the first set has not output a final result of the task within the timeout period, to start performing the task; and

the second set of computing devices is configured, after starting performing the task and before outputting a final result of the task, to determine whether the first set has output a partial or final result since the second set started performing the task, and, if so, to output a final result that is consistent with the partial or final result output by the first set, and, if not, to output a final result of the task determined by the second set.

From a second aspect, the invention provides a method for performing a task in a distributed computer system comprising a plurality of networked computing devices, the method comprising:

a first set of the computing devices starting to perform the task;

a second set of the computing devices, different from the first set, determining whether the first set has output a final result within a timeout period, and, in response to determining that the first set has not output a result within the timeout period, starting to perform the task; and

the second set of computing devices, after starting performing the task and before outputting a final result of the task, determining whether the first set has output a partial or final result since the second set started performing the task, and, in response to determining that the first set has output a result since the second set started performing the task, outputting a final result that is consistent with the partial or final result output by the first set, or in response to determining that the first set has not output a result since the second set started performing the task, outputting a final result of the task determined by the second set.

From a third aspect, the invention provides a computing device for use in a distributed computer system that comprises a plurality of networked computing devices, configured for performing a task, including a first set of the computing devices and a different, second set of the computing devices, wherein the computing device is configured for membership of the second set of computing devices, and is configured:

to determine whether the first set of computing devices has output a final result of the task within a timeout period, and, if the first set has not output a final result within the timeout period, to start performing the task; and

after starting performing the task and before outputting a result of the task, to determine whether the first set has output a partial or final result since the computing device started performing the task, and, if so, to output a result that is consistent with the partial or final result output by the first set, and, if not, to output a result of the task determined by the computing device.

From a fourth aspect, the invention provides computer software comprising instructions for execution by a computing device for use in a distributed computer system that comprises a plurality of networked computing devices, configured for performing a task, including a first set of the computing devices and a different, second set of the computing devices including said computer device, wherein the instructions, which executed by said computing device, cause the computing device:

to determine whether the first set of computing devices has output a final result of the task within a timeout period, and, if the first set has not output a final result within the timeout period, to start performing the task; and

after starting performing the task and before outputting a result of the task, to determine whether the first set has output a partial or final result since the computing device started performing the task, and, if so, to output a result that is consistent with the partial or final result output by the first set, and, if not, to output a result of the task determined by the computing device.

Thus it will be seen that, in accordance with embodiments of the invention, if the first set of computing devices fails to complete the task in a timely manner (e.g. because it is being blocked from doing so by a malicious actor), a second set of devices will take over and perform the task. In this way, such systems are resilient against faults that might otherwise prevent the system from generating a result at all. Furthermore, if the first set of devices does, belatedly, output a partial or final result before the second set of devices has completed the task, the second set then aligns its result with the result output by the first set—this reduces the risk of contradictory (i.e. inconsistent) final results being output by the two sets of computing devices.

The distributed computer system may be a decentralized computer system. The computer devices may be connected by a local area network or by a wide area network. They may be connected by the Internet.

The task may be any processing task. In some embodiments, the task is a decision task and the or each final result is a decision. The first set of computing devices may perform the task as a first distributed process, and the second set of computing devices may perform the task as a second distributed process. The first and second processes may differ in the input data that each has access to and/or in the processing operations that each performs. The first and second processes may be distributed across the respective sets of computing devices in any appropriate way. For each of the first and second sets, the computing devices of a set may process the same or different data from each other (e.g. each device may have access to a respective portion of a larger set of input data), and may perform the same or different processing operations from each other.

In some preferred embodiments, the first set of computing devices is configured, after starting performing the task but before outputting a final result (e.g. a committed decision), to determine whether the second set of computing devices has started performing the task and, if so, to output no result or to output a final result that is the same as a final result output by the second set of computing devices. This may further help avoid a situation in which conflicting results are published, which might otherwise arise, e.g. as a result of asynchrony and communication delays in the system.

The second set of the computing devices may be configured, upon determining that the first set has output a final result within a timeout period, not to start performing the task. It may output no result for the task. Alternatively, in this situation, the first set may output the final result output by the first set (i.e. echoing the same result).

If the first set has output no result or only a partial result within the timeout period, the second set will start performing the task.

In some embodiments, the first and second sets of computing devices may be configured such that, if the first set of devices outputs a final result, it does so faster than the second set of devices outputs a final result, under some or all conditions. This

may be true on average over all input conditions. This may be true assuming the same input data is processed by each of the first and second sets of computing devices.

The first and second sets of computing devices may be configured such that the second set of devices is more likely to output a final result (i.e. to complete the task) than the first set of devices. The second set may be configured to be more robust than the first set against a respective (malicious) computing device of the set attempting to block the output of a final result by the respective set.

The second set of device may be larger than the first set of devices. This may make it harder for a malicious actor to prevent the second set of devices from completing the task. The second set of devices may be arranged so as to always output a result (albeit potentially more slowly than the first set, when the first set does complete the task). However, the first set of computing devices may be configured such that, under some conditions, the first set of devices will never output a result. Because the second set of computing devices can act as a fall-back when the first set fails to output a timely result, embodiments of the system may always, or almost always, output a result even if the first set fails to do so (e.g. because it reaches a deadlock and is unable to confirm any particular decision value).

In some embodiments, the distributed computer system is a blockchain system storing a blockchain, and the computing devices are respective blockchain nodes. The blockchain may be configured to store cryptocurrency. The blockchain may be configured to store smart contracts. The system may be configured to distribute a candidate block of transaction data to each of the first set of devices. The transaction data may relate to cryptocurrency transactions or smart contract transactions or both. The candidate block may be created from a pool of unverified transaction data. The task may comprise determining whether to add a candidate block to the blockchain. The candidate block may be generated by one of the first or second set of computing devices, or it may be generated by a different blockchain node. Each set of computing devices may be a plurality of blockchain nodes that is configured to decide collectively whether to add the same candidate block to the blockchain.

When the task comprises making a decision, the first and second sets of computing devices may implement respective consensus mechanisms for making the decision.

These consensus mechanisms may be the same or different for the first and second sets. The first set may require at least a first threshold number of the first set of computing devices to agree before the first set outputs a final result (e.g. at least half). The second set may require at least a second threshold number of the second set of computing devices to agree before the second set outputs a final result (e.g. at least half). The second number may be greater than the first number. This may result in the second set issuing a decision more slowly on average, but make the second set more robust against faults or attacks that might otherwise prevent it from outputting a result.

If the first set outputs a final result, the second set may output a consistent result by outputting the same final result. If the first set outputs a final result before the second set starts performing the task (i.e. within the timeout period), then the second set may be configured to output the same final result (i.e. to echo the decision of the first set), or may be configured to output no result. In some embodiments, the first set of computing devices may be configured to output final results for tasks and never to output partial results. However, in other embodiments, the first set may be configured to output a partial result before outputting any final result. It may, in some situations, output a partial result and then fail to output any final result (e.g. if it reaches a deadlock between outputting the partial result and the final result). When the task is a decision task, the partial result may be a decision made by one or more (e.g. by a threshold number of) computing devices of the first set that has not (yet) been approved by the first set (i.e. is awaiting approval) according to a first consensus mechanism implemented by the first set. Such a decision by a subset of devices of the set may be referred to herein as a “pre-decision” for the set.

In some embodiments, the decision task may be associated with a set of decision values (e.g. two, three or more possible decision values). For example, a first decision value of the set may be to approve an identified candidate block for inclusion in a blockchain, and a second decision value of the set may be to reject the identified candidate block for inclusion in the blockchain.

The computing devices of the first set may be configured such that a partial result output by the first set identifies a decision value selected from the set of decision values for the task. In some embodiments, for the first and/or second set, the computing devices may be configured such that any individual computing device of the

set can propose a decision (e.g. by outputting a signed certificate asserting what the decision should be). The set may then be considered to have output a partial or final result once a threshold number of the computing devices of the set has indicated agreement with the proposed decision. In some embodiments, the result may be a partial result (i.e. a pre-decision) of the set at least until sufficient of the computing devices (i.e. at least a further threshold number) have received the partial result, whereupon the set confirms the partial result as a final result.

The devices of the first set may be configured such that any final result output by the first set must confirm the decision value identified by the partial result. In other words, once a pre-decision has been reached, this pre-decision commits the first set to either confirming this pre-decision as the final result or to not outputting any final result at all (e.g. if consensus cannot be reached within the first set of devices, perhaps due to the actions of a malicious device within the first set preventing the first set from transitioning from outputting a partial result to outputting a final result).

The second set of devices are preferably configured such that, if a partial result is output by the first set and the partial result identifies a decision value, the final result output by the second set must confirm the decision value identified by the partial result output by the first set. In this way, the final result of the second set may be consistent with any partial or final result output by the first set.

The first set and second set may contain at least one computing device in common. The first and second sets may intersect, or one of the sets may be a subset of the other set.

Each of the first and second sets of computing devices may comprise one or more "decider" computing devices configured to output a final result for the respective set of devices. A decider for the first set may implement a first consensus mechanism for the first set, and a decider for the second set may implement a second consensus mechanism for the second set. Each decider may be configured to collect partial results output by the computing devices of the set and to determine when it has collected a threshold number of consistent partial results (e.g. partial results that indicate agreement with a same decision value). There may be a single decider for

each set, or each of the plurality of computing devices of each set may be configured to act as a decider for the respective set.

The timeout period may be determined based on a time at which a final decision of an earlier task was output. For example, it may start at a time that a block (e.g. a most recent block) was added to a blockchain.

Each computing device of the second set may be configured to determine when the timeout period elapses.

In a first set of embodiments, the computer devices of the second set may do this by each computing device of the second set directly measuring the time period (e.g. using a clock or timer of the device).

In a second set of embodiments, each computing device of the second set may be configured to receive a signal indicating that the timeout period has elapsed (e.g. from one or more computing devices of the first set). Each computing device of the first set may be configured to determine when the timeout period elapses (e.g. using a clock or timer of the respective device). One or more, or each, of the computing devices of the first set may be configured, after determining that the timeout period has elapsed, to issue a timeout signal indicating that the timeout period has elapsed. These timeout signals may be issued for receipt by computing devices of the second set. The computing devices of the second set determining when the timeout period elapses may comprise each computing device of the second set detecting a timeout signal issued by one or more of the computing devices of the first set indicating that the timeout period has elapsed. In some embodiments, each of the second set of computing devices is configured to determine that the timeout period has elapsed only after detecting a timeout signal from more than a predetermined proportion (e.g. half) of the computing devices of the first set of computing devices.

The computing devices (of the first set or of the second set) may, in some embodiments, use respective clocks that are not all mutually synchronised. The timeout period may therefore be determined slightly differently by each respective computing device.

The system may comprise one or more further sets of the computing devices configured to start performing the task when respective predetermined conditions (e.g. timeout conditions) are met. Each may be configured to start performing the task in response to determining that a set of devices (e.g. the first or second set) has not output a result of the task within a respective timeout period.

Outputting a result may comprise publishing the result such that the result is available to a majority or all of the plurality of networked computer devices. It may comprise issuing a certificate, which may be cryptographically signed. Some or all of the computing devices may be configured to output a partial result as a respective partial certificate. A decider for a set of computing devices may be configured to output a final result for the set as a final certificate.

Each computing device may be a workstation, server, laptop, mobile telephone or any other computing device. It may comprise memory and one or more processors for executing software stored in the memory. Computer software as disclosed herein may be stored on a non-transitory computer-readable medium such as a magnetic or solid-state memory.

Features of any aspect or embodiment described herein may, wherever appropriate, be applied to any other aspect or embodiment described herein. Where reference is made to different embodiments or sets of embodiments, it should be understood that these are not necessarily distinct but may overlap.

BRIEF DESCRIPTION OF THE DRAWINGS

Certain embodiments of the invention will now be described, by way of example only, with reference to the accompanying drawings, in which:

Figure 1 is a schematic diagram of a blockchain computing system embodying the invention;

Figure 2 is a flow chart of a method, embodying the invention, for performing a task by a distributed system under a first set of conditions;

Figure 3 is a flow chart of the method for performing the task under a second set of conditions; and

Figure 4 is a flow chart of the method for performing the task under a third set of conditions.

DETAILED DESCRIPTION

Figure 1 shows a schematic representation a blockchain network 100 comprising a plurality of nodes 102 that can communicate with each other over the Internet to maintain a distributed blockchain. Each node 102 is a respective computing device, such as a workstation or a server, that contains memory and one or more processors executing software for performing operations to maintain the blockchain. Each node 102 stores a local version of the blockchain in its memory. At any instant these various copies of the blockchain should be very similar or identical. However, due to communication and processing delays, they may at times differ slightly, e.g. with regard to one or more of the most recently-added block or blocks. Although Figure 1 shows only nine exemplary nodes 102, it will be appreciated that the blockchain network 100 may include any number of nodes, which may be much great than nine. Nodes may join or leave the blockchain network 100 over time.

At least some of the nodes 102 are block-creating nodes and are arranged to receive transactions (e.g. currency transactions) and to store these in a local pool of unconfirmed transactions. These block-creating nodes are programmed to generate candidate blocks for inclusion in the blockchain based on transactions selected from the pool of unconfirmed transactions. There may be other nodes 102 that do not undertake block-creating activities, but nevertheless maintain local copies of the blockchain.

The blockchain network 100 implements a consensus mechanism for deciding which blocks to include in the blockchain. This mechanism requires a set of the nodes 102 to act as verifier nodes and to check any candidate blocks to verify that it meets predefined requirements for inclusion in the blockchain. It is possible for verifier nodes 102 to disagree regarding whether a particular candidate block should be included, e.g. because some of the nodes 102 may have access to information that is not known to others of the nodes 102, or due to latency issues in the network 100. The set of verifier nodes may implement a voting policy to decide whether a candidate block should be included. The result of decision may be published across the network 100 by the set of nodes—e.g. by way of a certificate issued by one or more of the nodes of the set. The nature of the voting policy, and a required size of the set of verifier nodes, may be parameters of the network 100, which may change under different conditions.

In some situations, a set of verifier nodes may collectively fail to reach a decision. Such a failure might, in some situations, be caused by a malicious actor causing a proportion of these nodes to behave badly—e.g. to try to vote in a candidate block that contains a malformed transaction, such as double-spending cryptocurrency. At times the nodes may enter a deadlock condition where no decision is published.

When a first set of the nodes, such as the first set 104 of three nodes 102b, 102c shown in Figure 1, fails to reach a timely decision on a candidate block within a timeout period in a first process, the task may be handed over to a larger set of nodes, such as the second set 106 of five nodes 102a, 102c shown in Figure 1, which may use a second process (which may be the same as the first process or different). The second set may be determined by the system 100 so as be larger than the first set. This may make it much less likely, or even practicably impossible depending on the size of the second set, that a badly behaving actor could control a sufficient proportion of the nodes 102 so as to be able to cause the second set to fail to reach a decision—i.e. to deadlock. The second set 106 may include one or more nodes 102c that are also in the first set 104. In some embodiments, the second set may include all the nodes of the first set—i.e. it may be a superset of the first set. It could include every available (i.e. online) node 102 of the network 100 that is capable of acting as a verifier node.

The second set of nodes may be determined and controlled by the blockchain network 100 to such that a decision will always be reached. A set of nodes may be referred to herein as "complete" if it is stochastically impossible for the set to fail to generate an answer. Whether a set of nodes is complete may depend on factors including how the set is chosen (e.g. its size); an underlying trust model; an upper bound on the probability p of any node of the set being bad or corruptible; and the type of decision required (e.g. a majority vote, or some other decision process).

However, it may be the case that the second set, although complete, will, on average, take significantly longer to reach a decision than the first set, when the first set does not deadlock (which should not occur often in normal usage). This approach of trying a smaller set of nodes first, then opening the task out to a larger set of nodes, can provide a desirable mix of efficiency and resilience.

Figures 2-4 show how a processing task can be carried out, by a distributed computer system embodying the invention, under three different sets of conditions. The system could be a blockchain system, such as the system 100 of Figure 1, with the task being to decide whether to accept or reject a candidate block. However, it could be any system for performing a task, especially one where it is important that an unambiguous result is always reached.

This particular example uses a single-stage decision-making process, in which a set of devices moves straight to outputting a final result once a consensus decision is reached. Further down, other embodiments are described that use a two-stage decision-making process, in which a set of devices first outputs a pre-decision (i.e. a partial result) which may then be configured as a final decision (i.e. a final result), assuming consensus can be reached.

The system includes two sets of processing devices or nodes, Set A and Set B. These sets could be disjoint, overlapping or nested. They could be the first set 104 and second set 106 of nodes 102 of Figure 1, or any other sets of processing devices in a distributed (e.g. decentralized) system.

The task is carried out cooperatively, in a distributed fashion, by respective software executing on the nodes, in accordance with a defined protocol (e.g. a blockchain protocol). It is possible, however, that some of the nodes may have been compromised by a malicious actor, or may be faulty, and so may be behaving in ways that are contrary to the defined protocol. The methods described can provide robustness even in such situations.

Figure 2 shows what happens when Set A publishes a timely decision. Figure 3 shows what happens when Set A publishes a late decision. Figure 4 shows what happens when Set A deadlocks and fails to publish any decision, and how Set B ensures that a decision is nevertheless published eventually.

In Figure 2, the nodes of Set A start performing 200 the task in a first distributed process. This start is signalled to one or more of the nodes of Set B, which start measuring a timeout period T. In this example scenario, Set A completes 202 the task

and publishes 204 a result, Result A, within the timeout period T. At the end of the timeout period T, Set B checks and detects 206 that a result has been published by Set A. Consequently Set B takes no further action. However, in some embodiments, Set B may optionally republish Result A.

In other embodiments, the timeout period T is measured by the nodes of Set A, rather than by the nodes of Set B. The nodes of Set A start measuring the timeout period T at the same time as they start performing 200 the task. In such embodiments, it is not necessary for the *start* of the timeout period T to be signalled to Set B (although it may still be).

Each node of Set A is configured to issue a timeout signal for detection by Set B upon determining that the timeout period T has elapsed. However, in this specific example scenario, Set A completes 202 the task and publishes 204 the result before expiry of the timeout period T, so Set A may not be required to signal the end of the timeout period T to Set B.

In Figure 3, the nodes of Set A start performing 300 the task in a first distributed process. This start is signalled to one or more of the nodes of Set B, which start measuring a timeout period T. In this example scenario, Set A completes 302 the task and publishes 304 a result, Result A, but only after the end of the timeout period T. At the end of the timeout period T, Set B checks and determines 306 that no result has yet been published by Set A. Consequently Set B starts performing 308 the task in a second distributed process. The parameters of the second process may differ from those of the first process—e.g. in the size of the sets and/or in how the nodes seek to reach consensus.

In alternative embodiments, in which the timeout period T is measured by the nodes of Set A rather than by Set B, each node of Set A issues a timeout signal upon determining that the timeout period T has elapsed. After detecting a timeout signal from Set A (e.g. from at least half, or another predetermined proportion, of the nodes of Set A), Set B determines 306 that no result has yet been published by Set A and consequently starts performing 308 the task.

After a time from starting the task (which may be longer than the timeout period T that Set A was expected to take), Set B completes 310 the task. However, before publishing a result from the second process, the Set B nodes check again whether a result has been published by Set A (i.e. a late result, after the timeout period T), and, in this example, they detect 312 that a result, Result A, has been published. Consequently, Set B disregards whatever result was determined by the second process and instead publishes Result A. This ensures that two contradictory results cannot be published for the same task.

In Figure 4, the nodes of Set A start performing 400 the task in a first distributed process. This start is signalled to one or more of the nodes of Set B, which start measuring a timeout period T. Alternatively, Set A may start measuring the timeout period T at the same time as starting performing 400 the task, as described above.

In this example scenario, Set A fails to complete the task (e.g. is unable to reach a decision) and enters a deadlock situation. The first process may eventually be aborted, but no result is ever published. At the end of the timeout period T, Set B checks and determines 406 that no result has yet been published by Set A. Consequently Set B starts performing 408 the task in a second distributed process. Again, the parameters of the second process may differ from those of the first process—e.g. in the size of the sets and/or in how the nodes seek to reach consensus. After a time from starting the task (which may be longer than the timeout period T that Set A was expected to take), Set B completes 410 the task. Before publishing a result from the second process, the Set B nodes check again whether a result has been published by Set A (i.e. a late result, after the timeout period T), and, in this example, they determine 412 that still no result has been published. Consequently, Set B publishes the result, Result B, that it determined by the second process. In some embodiments, Set A may optionally detect the result published by Set B and republish Result B.

In all these scenarios, the second process performed by Set B may be such that Set B will always reach a decision—i.e. Set B may be complete. Thus, this approach should always result in a decision being reached, while also avoiding the possibility of inconsistent results being published.

The approach described here may be applied to many types of distributed computing system that are configured to perform distributed processing tasks so as to attempt to find consensus about a decision or series of decisions, and not only to blockchain systems. Such decisions might be delegated to a subset of computing devices (i.e. nodes) of the system, or might be taken by the whole, or a fixed subset, of the community. In cases where a decision is not made by a subset of nodes, it can be advantageous to try another subset or the entire community. Some of the methods disclosed herein address the challenging of doing this cleanly, given that when decisions are made in decentralized systems (i.e. the timing of decisions) is not always apparent or obvious.

It is important to note that decisions in the context of decentralized systems can be hard to arrive at definitively. There may be a point in time where the decision is made, becoming inevitable, but, because of communication delays and the asynchronous nature of decentralized systems, it may be that only one or a few of the nodes becomes aware of this at this particular point in time, and perhaps none at all. Individual nodes may become aware at different times, e.g. through either creating or seeing a certificate. It can be desirable that, in any circumstances where a decision is possible, a result (e.g. a certificate) will emerge, and that contradictory results (e.g. certificate) will never emerge, as indeed at least some of the embodiments disclosed herein can ensure is the case.

A more formal understanding of some of the principles disclosed herein will be provided, based on the concept of a consensus machine, which achieves a clear definition of decisions and an operational semantics that gives a way of programming this type of decision with considerable generality. This allows the “compiling” of programs that are essentially sequential and represent the intended behaviour of a decentralized system, together with the trust model and strategy for working with it, into descriptions of agents that together produce a trustworthy implementation. Typically this might be the process of managing the building and verification of a blockchain or similar structures, implemented using a strategy of delegation to groups of nodes, such as nodes selected, by suitable criteria, to be verifier nodes for the system.

Any system running on a decentralized system that may have some misbehaving bad agents is bound to have complications when trying to understand it. Consensus machines are a mechanism for making such systems clear. Collections of agents can be created from which the desired program becomes emergent behaviour, tolerating the presence of bad agents. The formulation below is based on process algebra for understanding bodies of agents that run concurrently and interact by forms of synchronisation. In it, all the agents know the system description and the good agents do exactly what that requires of them. The synchronisations of this machine are certificates generated by the groups of nodes that are legitimately running, noting that we will ensure it impossible for any other to be started.

A consensus machine represents the operating system and flow control of a decentralized systems. The program running on a consensus machine has a number of layers. At the top layer is a sequential program that proceeds in steps, which take the form of certificates of agreement to perform the step. These steps are significant to the whole community, for example the choice of the next block in a blockchain. Those delegated such a job might gather and agree a set of information, developing the data structures of the system. The group delegated this decision might proceed in small steps which are local to individual groups of delegates as discussed below. These still require certificates of agreement. This program can emerge from the synchronisations of the lower levels leading to eventual agreement on the big steps.

At the next level down is the strategy for getting agreement on the next big step in this. When a step starts a set of nodes will be operational — either inherited from the previous step, by initial conditions if this is the first step, or by construction of that step. There will also be a strategy for making the decision when set of nodes fails. Typically this will be trying a succession of sets of nodes, potentially growing until there is a large set that is certain to deliver a verdict.

When passing decision making from one group of nodes (i.e. processing devices) to another, the transition might come because the first group has the evidence that it will not be able to decide, and passes the token across itself. Alternatively, almost certainly because bad agents fail to participate, it fails either to decide or see positively that it will not. In both cases care is taken that control will not be passed across when

some agents in the first are already committed, or at least that then the second produces the same decision.

This can be understood through the use of communicating sequential processes (CSP), which is a formal language for describing patterns of interaction in concurrent systems. CSP has a number of ways in which one process can pass control to another. The throw operator P □A Q runs like P until it throws an exception in the set A, which causes it to run like Q. On the other hand the interrupt operator P/\Q has P run, but if Q performs any visible action it takes over.

The more difficult of these cases is where one group of nodes is taking over from another by its own action. That is because if a group of agents decides to hand over, there will be sufficient agreement to do so, excluding any other action. Handing over to a second group means that the first group has not made the big step decision and furthermore no member of the group can legitimately decide it has as that would be inconsistent with the decision that it cannot make one.

On the other hand asynchrony means that a second group of nodes which is more independent can grow impatient with a first and decide to take over from it independently from some node in the first making a decision which allows it to generate a certificate for a big step decision. Embodiments may here insist that if any group takes over — directly or otherwise — from one that is capable of taking a decision, then it comes to the same answer. Essentially this means that whenever an incomplete process is coming close to making a big step decision — which is the unique possible outcome by some time — then the process flags this where others will see it before actually generating a certificate.

In the present description, this /\-like take over, which we will mainly expect to be a time-out, is limited to the case where the group taking over is complete. That means that if mechanism B takes over of its own volition from A, it can be relied upon to deliver a decision, and so there is not a need for a yet-lower level of machine needing to trace the actions of both A and B. In other words, in taking a decision, a community will start off using one set of nodes, which (if incomplete) can explicitly hand over to another set, and so on. If no decision is taken by some specified time (e.g. a defined timeout period), a complete set will take over.

Consider control belonging to a group G of nodes initially, which generates a certificate that it wants to hand over to a second group H of nodes, where there is freedom concerning the relationship (e.g. disjoint or superset) of G and H, and whether H is complete. This certificate prevents further meaningful progress in G and triggers the agents making up H to start or take over computation in the state determined by the certificate. A trace of synchronisation certificates of G is continued by this transfer certificate and the trace of synchronisation certificates that H (and its successors) generate. The set of nodes controlling the current computation is a function of the current trace.

In general it is desirable for the progress of the consensus machine to be visible to all. Any good agent (or perhaps a core subset with certainly some good members that everyone uses as a reliable core) should post certificates in strict order. The present description considers takeovers (as opposed to handovers) where the initiator of the takeover is complete. It can therefore be assured that a certificate will be forthcoming when justified, and the most obvious justification is the lack of a certificate by the time a decision is expected.

There is no practicable way that any agent can make its actions visible to all instantly in a decentralized system. There is a possibility that, where a complete group H has the job of taking over when nothing is seen on time from G, it might start doing so as a certificate emerges from G. At least some embodiments described here address this need for a way of H preventing G from further progress and for a way in which sufficient progress from G can prevent H from taking over, but while accepting a degree of a asynchrony. They do so by seeking to minimise the possibility of two decisions coming from G and H respectively, and to ensure that if this does happen then they represent the same decision.

It may be impossible to avoid the following: G generates a certificate for a decision after H starts and so eventually reaches a decision itself. In this case, embodiments may ensure that the same decision is reached by both.

H can take the form of a time-out. If an agent in H has not observed a decision by G by some time T then it will start its H program. The first action of H is to agree that it has started. Each action of G is augmented by the condition that H has not started.

Whenever the original G takes a final decision, the augmented one does not, rather it moves to a pre-final state which can agree to take that decision subject to H not having started. Thus G cannot take this decision until a decisive majority have reached the pre-final state.

When H starts up, its nodes seek to hear from G if they have reached the pre-final state, and treats this as input. If they have made a pre-decision D, H makes that decision. Note that in this state some node of G might also have created a certificate for D, so H following D avoids ambiguity. Similarly when H starts up it communicates this to G, which tests for the negation of this before progress is allowed. However again embodiments may need to allow for asynchrony.

In the model above, G should communicate to H when there is the risk a certificate might have emerged from it. Similarly H communicates to G when it starts to prevent G developing further and making a decision that it is not already committed to when H begins. This is not as simple as it may, due the networked distributed system being asynchronous and nondeterministic. The nondeterminism is that, if a small number of states of one machine have reached a state, there is no guarantee that the other will be able to see this yet.

What it means for one group of nodes to make progress is the existence of a certificate of this. The following analysis shows how such groups may communicate with each other. It assumes that appropriate and secure (i.e., unforgeable) items are written in some multiply redundant way, and read in such a way that the reader is sure it will have accessed some of these places. In this protocol, there may be three sorts of things in these slots:

1. a null value, which is assumed if nothing else valid is found;
2. the certificate for a decision D by G. Note that any certificate's validity is checkable, and its mere existence represents a state change in G; or
3. the indication that the background machine H has started. This may advantageously be represented by a certificate, which is unforgeable.

In all contexts where certificates are used, including here, they are preferably uniquely bound to the decision they are contributing to. In other words they securely record not only what was agreed but also tie it uniquely to its place so that it is impossible to forge, even knowing many similar certificates.

Two models of writing and reading are contemplated that might be in place for this. In one, each agent writes to a slot that it maintains, unique to it. When instructed to read, an agent looks at all these slots (i.e. across the whole of the group), returning a valid certificate value for the context if it finds one and null otherwise.

In the second, a group of agents (who may or may not be ones involved in other ways) is selected. These are written to and read from. What is important is a single good one is both written to successfully and read from. The combinatorics of this are rather different to those of dominated decisions, as it unnecessary to have a majority of good agents, merely the sufficient availability of one.

Some further exemplary embodiments will now be described, by way of CSP representations. These embodiments implement a two-stage decision-making in which a pre-decision (i.e. a partial result) is generated first. This commits the system to deciding on a particular decision value. If consensus is reached for this decision value, the system will then confirm it as the decision for the system (i.e. a final result).

The CSP representations below can be proved correct using a Failures/Divergence Refinement (FDR) tool. The first CSP abstraction models two consensus machines: G (assumed safe but cannot relied on to terminate) and a back-up H, connected only by decentralized stores where writing introduces a nondeterministic phase. The second CSP abstraction represents a more-detailed distributed model that demonstrates how the emergent behaviour of G and H arises due to the actions of individual nodes within each group.

First, some background is given, followed by the two CSP abstractions. In the following description, verifier nodes are referred to as “pickets”.

A unitary consensus machine

One of the main problems in designing a blockchain is devising how to select a unique successor for a given block; the initial (often termed genesis) block is pre-agreed between agents and assumed to exist, however there may be more than one plausible candidate for any subsequent block. This problem is often solved by a protocol that determines whether a block is final in blockchain terminology.

Typically, the finality of a block is determined by a universally known, though potentially randomly selected, committee of agents, which we call pickets, that engage in a protocol by which they reach a consensus on the successor of a given block. We call such a system composed of interacting pickets that solves the problem of determining the finality of a block a consensus machine. Since blockchains are systems intended to cope with adversarial behaviour (coming from untrusted parties), these machines are designed to tolerate a certain proportion of malign agents. That is, the expected overall behaviour emerges from the interaction of pickets in spite of possible misbehaviour by malign agents amongst them. The notions described here can also be applied to the problem of reaching consensus for more general distributed systems.

We first illustrate how a unitary consensus machine works, i.e., how a single set of pickets can interact to reach consensus. Later, we build on this illustration to propose our hierarchical consensus machine. The informal description that we provide here illustrates the mechanism used by the hierarchical protocol we propose later.

Let P be a set of pickets, D be a set of possible decision values that the pickets are trying to reach consensus on, and $M \subseteq P(P)$ the decision sets such that agreement by any set $m \in M$ commits the system to the agreed decision, where M is superset closed and contains P and $P(S)$ gives the power set of S. Broadly speaking, the unitary consensus machine works as follows. For a given run of this machine P, M, and D are fixed and well-known. Each picket $p \in P$ locally decides on a single value $v_p \in D$ and broadcasts this chosen value. We assume that pickets have well-known public keys as part of agreed cryptographic signature schemes so they can create unforgeable digitally signed messages. The set $m_{o,v}$ denotes the set of pickets that have chosen value v according to the messages received by observer o. If $m_{o,v} \in M$, observer o knows that the machine has decided on value v. In this description, we focus on a restricted scenario involving a single run of the machine, i.e., having pickets decide on

a value a single time. There is no issue in extending this for a series of decisions where each is properly made before the next one starts.

We note that since the evidence for a decision will be an agreed and signed decision by sufficient agents for some m ∈ M, no-one can dispute a properly formed one. Whatever decision is made is agreed with by at least one benign agent that follows all the rules: this will be a property of M. Well-behaved consensus machines should additionally respect two properties:

- Safety: For observers o_1, o_2 and v_1, v_2 ∈ D, if $\mathrm{m}_{\mathrm{o}_1,\mathrm{v}_1,}$ ∈ M and m_{o_2,v_2}∈ M, it should be the case that $v_1 = v_2$.
- Liveness: After observing the consensus machine run for stabilisation time t, an observer o is able to construct a set $m_{o,v}$ such that $m_{o,v}$∈ M.

Intuitively speaking, the safety property forbids the machine from deciding on two distinct values (on the same run), whereas the liveness property ensures that the machine eventually decides on a value. Note that the liveness property implicitly accounts for enough pickets agreeing on a given value but also for their decision being conveyed in a timely manner.

We assume that malign agents can deviate from the expected picket behaviour arbitrarily. For instance, they could send as their chosen value v_1 to observer o_1 while sending a distinct value v_2 to observer o_2 — this double choice is a common byzantine behaviour expected of such malign agents.

The safety property depends on M considering the benign and malign agents in P. For instance, if sets m_1, m_2∈ M are crafted so that there is no benign picket that is part of both m_1 and m_2, these two committees could decide on two distinct values on the same run. Thus, (i) any two sets in m_1, m_2 ∈ M must have an overlapping benign picket to achieve the safety property. We can show (i) by contradiction. Let us assume that sets m_{o_1,v_1}∈ M and m_{o_2,v_2}∈ M with $v_1 \neq v_2$ were constructed. Then, by (i), p′ ∈ m_{o_1,v_1} and p′ ∈ m_{o_2,v_2}∈, which implies that the benign picket p′ choose two values v_1 and v_2, a contradiction.

To ensure liveness, one should assume or enforce that: (a) there is some stabilisation time by which point messages from benign pickets are delivered and (b) a set m ∈ M

of benign pickets chooses the same value (in time for stabilisation); the stabilisation time is required to move away from impossibility results. While (b) ensures a decision is made, (a) ensures that an observer can witness this decision. For our minimalist unitary consensus machine present in this section, we assume that such a set m exists as pickets are making their choice. In practice, however, if no set $m \in M$ could be constructed — when, for instance, pickets choose different values — the protocol would have a recovery mechanism by which pickets would choose another value to try and build such a m; the protocol would be constructed so that pickets converge into an agreed value after some time.

It is important to understand the dichotomy between safety and liveness in the setting we study: one can be more tolerant of malign pickets' involvement when crafting an M that is safe but not live as opposed to one that is safe and live; this observation follows from properties (i) and (b). There are decision sets M that abide by property (i) and yet cannot satisfy (b). For instance, we could have an M that abide by (i) but all its member include a malign picket. In these cases, the participation of benign nodes in the members of such an M ensure decisions are safe. However, the presence of malign pickets may cause a decision to never be reached as they can refuse to participate in the consensus protocol. This observation is one of the main principles guiding the design of our hierarchical consensus machine.

In the context of blockchains, a consensus machine is meant to determine the true/canonical chain by repeatedly picking successor blocks — and pruning the block tree in the process. These blocks, and the transactions that they contain, represent transitions in the state of the blockchain. They can account, for instance, for a transfer of digital currency or the execution of some code (i.e. via a smart contract). Thus, assuming that these transactions are deterministic, the consensus machine also determines the canonical sequence of states of the blockchain.

Blockchains are frequently set up with incentive and penalty structures that are designed to persuade the malign agents to follow the rules. We categorise malign behaviour as follows:

1. Overt malign behaviour. Making contributions to the central discussions and protocols of a chain or other decentralised system that will be seen and

recognised as malign. Unless this wins votes or similar, it will quickly be recognised and the perpetrator punished.

2. Covert malign behaviour. Producing non-compliant structures that are kept hidden and only perhaps revealed later. For example developing a fork alongside the true chain.
3. Non-participation. Failing to make contributions that are expected of a good agent and thereby denying some correct action the majority it needs. The main issues with this is that it is harder to penalise because a good agent may encounter communication failures, a phenomenon that can also mean confusion about how an apparently non-participating agent should be interpreted.

Gossiping assumptions may be made about communications in blockchains to resolve such confusion.

The sorts of incentive structures implemented by blockchains are another important factor that guided the design of our hierarchical consensus machine. In particular, non-participation failures may cause the need to transfer control from one unitary consensus machine to another, in order to achieve overall liveness.

Stochastic decisions

The security analysis of blockchains is usually predicated upon some assumed distribution of malign agents. So, we use probability to assemble sets of pickets and produce decision sets M. In this section, we discuss a central case of how this can support the picketing model. We assume that pickets are drawn from an agent population U where the probability that a randomly chosen agent is benign is p, and that they are selected independently and randomly from U so that the number of benign and malign pickets that make any decision set is governed by a binomial distribution, that is, $\binom{n}{k}p^k(1-p)^{n-k}$ gives the probability of having k benign agents when selecting n agents from U. Given this assumption, it is relatively easy to compute how likely it is that at most r out of n picket selections are benign: $F(p,n,r) = \sum_{i=0}^{r}\binom{n}{i}p^i(1-p)^{n-i}$.

Based on these s, we propose the idea of *stochastic impossibility*: an event so unlikely that in the whole history of a system it is very unlikely that one will happen, to the

extent that it can be disregarded. This concept is parameterised by a *insignificance threshold* ϵ and an event that happens with probability $\xi \leq \epsilon$ is stochastically impossible. One might regard a one-in-a-million chance as small enough, but if many (say a million) choices are going to be made a year (approximately one every 30 seconds) it is clearly is not enough if a single one can corrupt a system. We believe that the $\epsilon = 10^{-18}$ is a reasonable starting point; in terms of the normal distribution, this value is close to 9σ ($\approx 10^{-19}$), where σ is the standard deviation, namely, the cumulative probability from $\mu + 9\sigma$ to infinity, where μ is the mean. This sort of σ-multiplier analysis is used in finance to model risk, and is justified as a consequence of the probabilistic laws of large numbers.

We can now understand how to create the decision thresholds M described earlier. In a population U of agents each with independent probability p of being benign, a randomly drawn (with replacement) sub-multiset of pickets $P \subseteq U$ is said to have (stochastically certainly) at least k+1 benign agents if $F(p, k, |P|) < \epsilon$; this inequality means that having at most k benign agents is stochastically impossible. For fixed p, k, and ϵ, we can calculate the smallest |P| so that at least k + 1 agents are benign; let us call this threshold value $td(p, k, \epsilon)$. Given that a multiset of pickets P where $|P| = td(p, k, \epsilon)$ has at least k + 1 benign agents, any sub-multiset $m \subseteq P$ such that (1) $|m| \geq |P| - (k + 1) + b$ includes at least b benign agents.

To achieve safety via (i), we should have more than half of the k +1 benign agents in any $m \in M$. So, by using $b = k/2 + 1$ in (1), we have that $|m| \geq |P| - k + \lceil k/2 \rceil$, where k/2 is integer division. Therefore, for $M = \{m \subseteq P \mid |m| \geq |P| - k + \lceil k/2 \rceil\}$, we have that property (i), and safety, is satisfied, modulo stochastic certainty.

To achieve liveness via (b), we should have (2) $|m| \leq k + 1$, namely, at least a decision set that requires (modulo stochastic certainty) only the participation of benign agents for agreement. Thus, to have safety and liveness, one should satisfy (1) and (2). The inequality (I) $|P| \leq \lfloor 3k/2 \rfloor + 1$ should be satisfied in order to ensure both (1) and (2). This inequality gives the bounds that are usually referred to in consensus literature.

The table below illustrates some examples of calculation for the largest k such that $F(p, k, n = |P|) < \epsilon$, for some values of n (number of selected agents) and p (benignity probability) and where is fixed $\epsilon = 10^{-18}$. The p values were chosen with the consensus

bounds of > 2/3 in mind. This calculation is analogous to the one presented. Entries in the top left (denoted by '*//*') are where even seeing all agents agreeing does not prove this, as it is deemed possible that all the agents are malign, namely, for these values of p, n, and ϵ, there is no k such that $F(p, k, n) < \epsilon$. In cells containing asterisks (*), we have that k and n satisfy (I). We can achieve safety for all but the '//' cells in the upper left corner. However, safety and liveness can only be achieved for the asterisked cells in the right bottom corner. This pattern illustrates that achieving both safety and liveness requires larger sets of pickets and decision sets in comparison to achieving safety alone. For example, with p = 0.95 and n = 50, we have that k = 25. So, we have at least 26 benign agents amongst the 50 randomly and independently selected. Thus, to ensure safety, we can choose decision sets $m \subseteq P$ such that $|m| \geq 38$. Since (I) does not hold for n = 50 and k = 25, we cannot obtain safety and liveness. On the other hand, for p = 0.95 and n = 100, we have that k = 66, in which case (I) holds. For this case, we can have decision sets $m \subseteq P$ such that $|m| \geq 67$ to achieve liveness and safety. Smaller pickets and decision sets should also allow for more efficient agreement given that fewer agents need to actively take part in the protocol; this principle was one of the main drives in designing our hierarchical consensus machine.

0.66	*//*	*//*	0	3	6	14	22	70	177	290	525
0.75	*//*	0	3	7	12	22	32	91	218	351	624
0.8	*//*	1	5	10	16	27	39	104	243	387	***682***
0.85	*//*	3	8	14	20	33	46	118	***269***	***425***	***742***
0.9	0	6	12	19	26	40	55	***134***	***298***	***466***	***807***
0.95	3	10	17	25	33	50	***66***	***153***	***331***	***512***	***878***
p/n	20	30	40	50	60	80	100	200	400	600	1000

Usually our systems do rely on decisions being made, and usually the systems are more efficient if they can persuade malign participants to contribute mostly as though they were good. Indeed for a consensus machine with a smaller set of pickets to deliver results, this may be necessary. To achieve this they should have three things: firstly strong incentives on agents not to misbehave and to participate constructively, secondly a decision making mechanism that prevents the malign from inducing a bad decision, and thirdly a fallback mechanism that can force correct decisions (i.e. is both safe and live) when needed, all be it at the cost of lower efficiency. The last of these should convince opponents that they will not be able to permanently disrupt the

system. The worst they can achieve is complication and delay. One cannot reasonably prevent the malign from covert mischief, but overtly saying the wrong thing or not doing what they are meant to will attract penalties and bans. The main motivation for the hierarchical consensus machine idea introduced next is providing the required fallback mechanism. It allows us to initiate a decision on the assumption that (most of) the malign agents participate normally in the knowledge that the carefully-picked (safe) decision sets will prevent a bad decision from being made; the (live) fallback allows a decision to be forced even when malign agents do not participate.

Formalising hierarchical consensus machines in CSP

We have already described how a consensus machine proceeds when it consists of a single set of pickets synchronising in a rather abstract sense. We have also described how to pick decision sets so that one can achieve safety and liveness using a type of stochastic reasoning. In this section, we present a hierarchical consensus machine, let us call it HM, that is in itself a combination of two (sub-)consensus machines, let us call them, G and H. The machine G is safe and efficient, whereas H is less efficient but it is both safe and live; as explained in the previous section the difference in efficiency comes from the size of picket and decision sets that are necessary to achieve these properties. In achieving safety without liveness, G can enter a situation very similar to the well-known phenomenon of deadlock, when malign agents refuse to take part and agree on a value. Deadlock is not normally an acceptable behaviour of a complete system, and certainly not in a blockchain. We propose a way to recover from such a deadlock in G by letting H take over. Specifically, we show how control of a decision-making procedure can be handed from one machine to the other. Despite G not being live, HM still is so thanks to H and the handover protocol we propose.

When passing decision making from G to H, the transition might come because the agents in G have the evidence that G will not be able to decide, or because malign agents in G fail to participate — in the latter case, G will not reach a decision but its agents are unable to determine that it will not. In both cases, we need to be careful that control will not be passed to H when some agents in G are already committed to a value, or at least that, in this case, H decides on the same committed value. So, our protocol does not prevent H and G both issuing decisions, but ensures that if they do, they are the same. The more difficult of these cases is where the pickets in H take over on their own initiative. That is because if G's agents themselves decide to hand

over, it will be because there is agreement to do so. Handing over to H means that G has not made the decision, and none of G's agents can validly believe it has, as that would be inconsistent with the agreement to hand over. When taking over from G, the H process does not have an immediate global effect on all the agents of G, so a decision may still be made later by G.

Our formulation is inspired by the large body of work on process algebra: understanding bodies of agents that run concurrently and interact by forms of synchronisation. There is an interesting analogy here with process algebra. CSP, particularly in later versions, has a number of ways in which one process can pass control to another. The throw operator P [| A |> Q runs like P until it throws an exception in the set A, which causes it to run like Q. On the other hand the interrupt operator P /\ Q has P run, but if Q performs any visible action it takes over.

We present and formalise in CSP two models for HM. The abstract model represents the behaviour of each machine G and H as a single CSP process. It abstractly depicts what is the expected emergent behaviour from their respective implementation each of which as an interactive distributed set of pickets. The main step of this abstraction is that the component consensus machines G and H are deemed to take an action only once there is agreement (in the sense we have already discussed) on the action. The distributed model, on the other hand, demonstrates precisely how the emergent behaviour of each machine can be realised in terms of such a set of pickets.

In other words the abstract model describes how we expect the protocol to work in every implementation, but the way in which the sequential processes it contains are implemented by decentralised collections implemented by G and H are not laid down. The distributed model illustrates one way of realising this. The protocol we present here has much in common with mutual exclusion. We want to prevent something akin to a race condition. An obvious question is whether we could use a simple mutex between G and H and only allow one to make the decision. The answer is no: it is part of the make-up of G that it can deadlock at any time. If it were to seek the right to make the decision — via the shared mutex — but then deadlock, then HM would deadlock too; contrary to our specification.

First CSP Model – An Abstract Model

In the abstract model, each of G and H is modelled as a single CSP process, and they communicate via shared storage locations each of which is also represented as a CSP process and each machine has two locations it can write to. Intuitively speaking, machine G comes to a decision in a two-step process. It first commits to (i.e. pre-decides on) a value by writing it on its first location and then it decides on this value by writing to its second location. Before these writes it checks whether H has started by looking for a started signal written to H's first location. If at any point it detects that H has started, it stops by choice. After a timeout has elapsed, H starts. It initially checks whether G has come to a decision already. If so, it reaffirms that decision. Otherwise, it signals it has started its decision making process by writing a started signal value on its first storage location. If no value has been committed to by G at that point, H proceeds to make its own decision. Otherwise, again, it just echoes G's decision.

Machines G and H rely on storage locations to communicate and convey (pre-) decisions. The datatype values denotes the possible values stored in these locations: D1 and D2 are (pre-)decisions whereas quiet, start and null denote machine statuses. Locations are identified by elements in location. Locations 1 and 3 are controlled (i.e. written) by machine G whereas 2 and 4 are controlled by machine H. Channels read, write1 and write2 are used to manage locations whereas stepG, stepH, and timeoutstep denote internal actions of these machines. Finally, channel decision is used to communicate (pre-)decisions made by them.

```
datatype values = quiet | started | D1 | D2 | null
locations = {1..4}
channel read, write1: locations.values
channel stepG, stepH, timeoutstep
channel decision:{1,2}.{D1,D2}
```

The storage locations are defined by the following two processes. Writing to and reading from these locations are not atomic events. When a value y is written to location i (via write1.i?y) the storage goes into a non-deterministic state in which it allows for a read to retrieve the old value x. The event write2.i signals to this location that the value y has been properly written at which point reads deterministically return y. This non-determinism captures (i.e. abstract) the asynchrony of distributed system:

the write begins when the decision is known somewhere and it ends when it is known at most of the network.

```
Store(i,x) = read.i!x -> Store(i,x)
      [] write1.i?y -> StoreND(i,x,y)

StoreND(i,x,y) = (read.i.x -> StoreND(i,x,y)
            |~| read.i.y -> StoreND(i,x,y))
      [] write2.i -> Store(i,y)
```

We abstract away all activities of G and H except the steps they need to make to record the decision they make and the steps they need to record and coordinate it. For modelling purposes we assume here that G makes decision D1 and H makes D2 unless it is forced to follow G's decision because it cannot be sure G will not make a decision.

The machine G's behaviour is defined by the following CSP processes, with initial state given by G. As G0, it reads the status of machine H via location 2. If H has started already, it stops. Otherwise, if H is quiet, as process G1, it signals a pre-decision on value D1 by writing it to Location 1. If H is still quiet at that point, it consolidates this pre-decision with event write2.1. As process G2, it reads the status of H for the last time, before issuing a final decision on D1 as process G3. Note that the parallel combination of G0 and CHAOS in G captures G's incompleteness by allowing it to deadlock at any point.

```
G0 = (read.2.quiet -> stepG -> G1
[] read.2.started -> STOP)

G1 = write1.1.D1 ->
      (read.2.quiet -> write2.1 -> stepG -> G2
      [] read.2.started -> STOP)

G2 = read.2.started -> STOP
      [] read.2.quiet -> stepG -> G3
```

```
G3 = decision.1.D1 -> write1.3.D1 -> write2.3 -> STOP

G = G0 [|Events|] CHAOS(Events)
```

The machine H's behaviour is defined by the following CSP processes, with initial state given by H. As H, it reads whether machine G has come to a decision by reading Location 3. If it detect a decision, it re-asserts this decision by writing D1 to Location 4. Otherwise, it interprets that a timeout has occurred and it moves on to make its own decision. As H1, it signals that it has started its decision making process by writing started to Location 2. As H2, it checks whether machine G has started at all. If it has, H re-asserts the pre-decision made by G — i.e., by writing D1 to Location 4. Otherwise, it proceed by making its own decision by writing D2 instead. Both of these decisions are captured by process H3.

```
H = read.3.null -> timeoutstep -> H1
      [] read.3.D1 -> write1.4.D1 -> write2.4 -> STOP

H1 = write1.2.started -> write2.2 -> stepH -> H2

H2 = read.1.null -> stepH -> H3(D2)
      [] read.1.D1 -> H3(D1)

H3(d) = decision.2.d -> write1.4.d -> write2.4 -> STOP
```

The hierarchical consensus machine behaviour is given by System. Note how machines H and G are interleaved in Machine and they rely on storage locations in Locations to interact as we discussed.

```
Locations = Store(1,null) ||| Store(2,quiet)
                  ||| Store(3,null) ||| Store(4,null)

Machines = G ||| H

System = Machines [|{|read,write1,write2|}|] Locations
```

We expect this abstract hierarchical consensus machine to be safe and live. By safe, we mean that if it comes to a decision, it decides on a single value, that is, each machine might even come to their own decision but their value must match. By live, we mean that System must not deadlock before a decision is made. We capture these two requirements by a refinement expression in CSP's stable failures model as follows.

```
Decisions = {write1.3.d, write1.4.d | d <- {D1,D2}}
Decision1 = {write1.3.d, write1.4.d | d <- {D1}}
Decision2 = {write1.3.d, write1.4.d | d <- {D2}}

DSystem = System \ diff(Events,Decisions)

Spec =(|~| x:Decision1 @ x -> CHAOS(Decision1))
       |~|
       (|~| x:Decision2 @ x -> CHAOS(Decision2))

assert Spec [F= DSystem
```

The refinement expression is built around decision events: all the decision events are members of Decisions, the decision events for value D1 are members of Decision1, and the events for value D2 are in Decision2. The specification process Spec allows a decision to be made on D1 and D2 initially. Once such a decision is made, only events deciding on that value are allowed be performed. Note that this process is not allowed to deadlock initially. Thus, the proposed refinement expression ensures that the behaviour of the system when projected onto decision events — given by DSystem — offers some decision event initially and stick to that decision value subsequently. We have used FDR to validate this refinement expression.

Second CSP Model – A Distributed Model

The abstract model is useful from an analysis perspective: one can examine the handover protocol itself while not needing to examine implementation of each machine as a collection of interactive agents and issues arising from such an implementation. Instead, issues with the handover protocol itself can be identified and fixed. We can then argue either that a given approach to building the individual machines G and H

will meet this model by construction, or test it by building a more detailed, distributed model in CSP for FDR.

In our model, each machine is a distributed system implementing a protocol that attempts to reach consensus in the presence of Byzantine agents. Intuitively speaking, our hierarchical machine works as follows. Machine G starts and tries to come to a decision on a unified value. After some appropriate amount of time — enough to allow G to come to a decision if agents can agree on a value — machine H starts. It checks whether machine G has committed to a value, i.e., it has pre-decided on it but might not have gathered enough evidence to properly decide on it. If so, machine H decides on that value. Otherwise, the agents in H are free to choose a value of their own. Like the abstract model, these machines communicate local decisions via storage locations.

Our more detailed CSP model is parameterised by some global functions. VALUES gives the universe of decision values, and NODES are the agent identifiers. For machine m, N(m) gives its number of agents, MNODES(m) are its agent identifiers, THRESHOLD(m) gives the level of agreement (i.e. how many agents) that is required for reaching consensus, G(m) gives the number of good agents, with GOOD(m) and BAD(m) identifying the good and malign agents in the machines, respectively. In the following, we describe in detail our CSP model.

```
datatype MACHINES = g | h

channel value : MACHINES.NODES.VALUES
channel prewrite, write : MACHINES.NODES.MACHINES.NODES.VALUES
channel setup_prewrite, setup_write : MACHINES.NODES.VALUES
channel decision : MACHINES.VALUES
channel decide : MACHINES.NODES.VALUES
channel timeout : MACHINES.NODES.MACHINES.NODES
channel end_round
```

We use event value.m.n.v to represent that the agent n in machine m has chosen as its decision value v. The event setup_prewrite.m.n.v is used to signal that agent n in machine m has pre-decided on value v (i.e. this agent is proposing this value as a

proposed decision for the whole machine), while the event prewrite.m.n.mm.nn.v is used to communicate to agent nn in machine mm that agent n in machine m has pre-decided on (i.e. is proposing) value v. The event setup_write.m.n.v is used to signal that agent n in machine m has decided on value v, while the event write.m.n.mm.nn.v is used to communicate to agent nn in machine mm that agent n in machine m has decided on value v. The event decision.m.v is used to signal that machine m has decided on value v, whereas decide.m.n.v conveys that agent n in machine m has (locally) decided on value v. The event timeout.m.n.mm.nn denotes that agent nn in machine mm timed out when trying to read the decision from agent n in machine m. The event end_round is a modelling device used to signal that machine G has had enough time to come to a decision and that machine H is now taking over.

```
EmptyPreWriteLocation(n,m) =
      setup_prewrite.m.n?v -> FullPreWriteLocation(n,m,v)

FullPreWriteLocation(n,m,v) =
      prewrite.m.n?mm?a:MNODES(mm)!v -> FullPreWriteLocation(n,m,v)
```

The process EmptyPreWriteLocation(n,m) is a storage location that stores the pre-decision of agent n in machine m; each agent has such a location that it controls. It is a single-write multiple-reads one-place buffer.

```
EmptyWriteLocation(n,m) =
      setup_write.m.n?v -> FullWriteLocation(n,m,v)
      []
      timeout.m.n?mm?a:MNODES(mm) -> EmptyWriteLocation(n,m)

FullWriteLocation(n,m,v) =
      write.m.n?mm?a:MNODES(mm)!v -> FullWriteLocation(n,m,v)
      []
      decide.m.n.v -> FullWriteLocation(n,m,v)
```

The process EmptyWriteLocation is also a storage location that behaves similarly to the previous one. It stores decisions instead of pre-decisions. Moreover, it offers a timeout event if the location is empty — it allows agents reading from it to experience a

timeout — and it uses the decide event to communicate the local decision of this agent.

```
GNode(n) =
      value.g.n?v -> setup_prewrite.g.n.v ->
      if v == 0 then PreWrite(n,g,{n},1,0,0)
      else if v == 1 then PreWrite(n,g,{n},0,1,0)
      else PreWrite(n,g,{n},0,0,1)
```

The control behaviour of agent n in machine G is given by process GNode(n). We design the agents so that they choose their local decision value independently (captured by event value) but they will come together, or not, to certify a unified decision. Once a value is chosen, it is written to the agent's pre-decision storage (via event setup_prewrite)—i.e. as a partial result for the machine.

```
PreWrite(n,m,vs,c0,c1,c2) =
      (prewrite.m?a:diff(MNODES(m),vs)!m.n?v ->
      if v == 0 then PreWrite(n,m,union({a},vs),c0+1,c1,c2)
      else if v == 1 then PreWrite(n,m,union({a},vs),c0,c1+1,c2)
      else PreWrite(n,m,union({a},vs),c0,c1,c2+1))
[]
      (timeout.m?a:diff(BAD(m),vs)!m.n ->
            PreWrite(n,m,union(vs,{a}),c0,c1,c2))
[]
(vs == MNODES(m) &
      if c0 >= THRESHOLD(m) then setup_write.m.n.0 -> EndOfRound
      else if c1 >= THRESHOLD(m) then setup_write.m.n.1 ->
            EndOfRound
      else if c2 >= THRESHOLD(m) then setup_write.m.n.2 ->
            EndOfRound
      else EndOfRound)

EndOfRound = end_round -> SKIP
```

The PreWrite process describes how an agent reads the pre-decisions of other agents (i.e. other agents' proposed decisions) in order to come to its own local decision. Once the agent has received a proposed decision or a timeout from all nodes, it goes on to either locally decide on a value or to conclude the decision making process without deciding on a value. If it has seen enough proposed decisions supporting value v — for instance, for v == 0, this is captured by condition c0 >= THRESHOLD(m) — the agent locally decides on v, writing this value to its decision storage location (via event setup_write). Note how the agent only accepts timeouts from malign agents; we assume that good agents deliver messages reliably and in a timely way. The process EndOfRound signals that machine's G time to come to a decision has elapsed, at which point, the agent terminates.

```
GGoodNode(n) = (GNode(n) [|{|setup_prewrite, setup_write|}|]
      (EmptyWriteLocation(n,g) ||| EmptyPreWriteLocation(n,g)))
```

A benign agent in machine G is a parallel process — given by process GGoodNode — that combines its storage locations and its control behaviour.

```
GoodAlpha(n,m) =
      Union({{| value.m.n, setup_prewrite.m.n, setup_write.m.n,
            decide.m.n, prewrite.m.n.mm.a, prewrite.mm.a.m.n,
            timeout.mm.a.m.n, timeout.m.n.mm.a, write.m.n.mm.a,
            write.mm.a.m.n, end_round | mm <- MACHINES,
            a <- MNODES(mm), (a != n or mm != m) |}})
```

GoodAlpha(n,m) gives the alphabet of the benign agent n in machine m.

```
HNode(n) =
      end_round ->
            Reader(n,{},0,0,0)

Reader(n,vs,c0,c1,c2) =
      (write.g?a:diff(MNODES(g),vs)!h.n?vv ->
            setup_prewrite.h.n.vv ->
            if vv == 0 then setup_write.h.n.0 -> EndOfRound
```

```
            else if vv == 1 then setup_write.h.n.1 -> EndOfRound
            else setup_write.h.n.2 -> EndOfRound)
[]
(timeout.g?a:diff(MNODES(g),vs)!h!n ->
      Reader(n,union(vs,{a}),c0,c1,c2))
[]
(vs == MNODES(g) &
      value.h.n?vv -> setup_prewrite.h.n.vv ->
            if vv == 0 then PreWrite(n,h,{n},1,0,0)
            else if vv == 1 then PreWrite(n,h,{n},0,1,0)
            else PreWrite(n,h,{n},0,0,1))
```

The control behaviour of a benign agent in machine H is given by process HNode(n). The initial end_round event and the requirements that we impose on the way in which agents synchronise on this event means that the agents of machine H only start after the agents of machine G have finished with their decision making interactions. This behaviour captures the assumption that agents have a reasonably synchronised clock and that they can come to a decision within a bounded time frame.

Once started, the agent's control behaviour in machine H is given by Reader. This process reads the local decisions made by agents in G. If one of them has decided on a given value — which means that machine G has committed to that value — we require that the agent in H decide on the same value. This behaviour ensures that if both machines come to a decision, they must agree on their decided value.

If no agent of G has decided on a value, the agents in H are free to choose their local decision values, and they move on to behave like process PreWrite to try and come to a unified decision as already mentioned.

```
HGoodNode(n) = (HNode(n) [|{|setup_prewrite, setup_write|}|]
      (EmptyWriteLocation(n,h) ||| EmptyPreWriteLocation(n,h)))
```

Similar to benign agents in G, a benign agent in H is a parallel combination of its control behaviour and storage locations as per process HGoodNode.

```
BadNode(n,m,c0,c1,c2) =
      timeout.m.n?mm?a:GOOD(mm) -> BadNode(n,m,c0,c1,c2)
      []
      (STOP
      |~|
      (prewrite.m.n.m?a:diff(MNODES(m),{n})?v ->
            BadNode(n,m,c0,c1,c2)
[]
prewrite.m?a:diff(MNODES(m),c0)!m.n.0 ->
      BadNode(n,m,union(c0,{a}),c1,c2)
[]
prewrite.m?a:diff(MNODES(m),c1)!m.n.1 ->
      BadNode(n,m,c0,union(c1,{a}),c2)
[]
prewrite.m?a:diff(MNODES(m),c2)!m.n.2 ->
      BadNode(n,m,c0,c1,union(c2,{a}))
[]
card(c0) >= THRESHOLD(m) &
      (write.m.n?a.b!0 -> BadNode(n,m,c0,c1,c2)
      [] decide.m.n.0 -> BadNode(n,m,c0,c1,c2))
[]
card(c1) >= THRESHOLD(m) &
      (write.m.n?a.b!1 -> BadNode(n,m,c0,c1,c2)

[] decide.m.n.1 -> BadNode(n,m,c0,c1,c2))
[]
card(c2) >= THRESHOLD(m) &
(write.m.n?a.b!2 -> BadNode(n,m,c0,c1,c2)
[] decide.m.n.2 -> BadNode(n,m,c0,c1,c2))))
```

The malign agent n in machine m is modelled by process BadNode(n,m). These agents can exhibit Byzantine behaviour but they are not allowed to behave completely arbitrarily: there are still some actions which these adversaries cannot perpetrate against benign agents. For instance, it can only offer event decide if it has gathered enough support for the corresponding decision — i.e., it cannot create a spurious local

decision. This abstraction accounts for the following behaviour: a local decision by an agent should be associated with enough supporting evidence — in the form of pre-decisions — which are cryptographically signed by the agents generating that evidence. We assume malign agents cannot break cryptographic primitives and, thus, they cannot forge signatures by other agents. On the other hand, malign agents can pre-decide on more than one value, or even refuse to serve a request for a (pre-) decision.

```
BadAlpha(n,m) = {| prewrite.m.n.mm.a, prewrite.mm.a.m.n,
      write.m.n.mm.a, write.mm.a.m.n, decide.m.n, timeout.m.n.mm.a
      | mm <- MACHINES, a <- MNODES(mm), (a != n or mm != m) |}
```

The alphabet of malign agent n in machine m is given by BadAlpha(n,m).

```
AlphaBadNodes(m) = Union({BadAlpha(i,m) | i <- BAD(m)})
BadNodes(m) = || i : BAD(m) @
      [BadAlpha(i,m)] BadNode(i,m,{i},{i},{i})

AlphaGoodNodes(m) = Union({GoodAlpha(i,m) | i <- GOOD(m)})
GGoodNodes = || i : GOOD(g) @ [GoodAlpha(i,g)] GGoodNode(i)
GNodes = GGoodNodes [ AlphaGoodNodes(g)
      || AlphaBadNodes(g) ] BadNodes(g)

HGoodNodes = || i : GOOD(h) @ [GoodAlpha(i,h)] HGoodNode(i)
HNodes = HGoodNodes [ AlphaGoodNodes(h)
      || AlphaBadNodes(h) ] BadNodes(h)

Nodes = GNodes [union(AlphaGoodNodes(g),AlphaBadNodes(g))
      || union(AlphaGoodNodes(h),AlphaBadNodes(h))] HNodes
```

The processes GNodes and HNodes capture the behaviour of machines G and H, respectively, whereas Nodes captures how they interact to implement the handover protocol. In these processes, the appropriate agents run in parallel and they are required to synchronise on shared events.

```
Decider(m,c0,c1,c2) =
      decide.m?a:diff(MNODES(m),c0)!0 ->
Decider(m,union({a},c0),c1,c2)
      []
      decide.m?a:diff(MNODES(m),c1)!1 ->
Decider(m,c0,union({a},c1),c2)
      []
      decide.m?a:diff(MNODES(m),c2)!2 ->
Decider(m,c0,c1,union({a},c2))
      []
      card(c0) >= THRESHOLD(m) & decision.m.0 -> Decider(m,c0,c1,c2)
      []
      card(c1) >= THRESHOLD(m) & decision.m.1 -> Decider(m,c0,c1,c2)
      []
      card(c2) >= THRESHOLD(m) & decision.m.2 -> Decider(m,c0,c1,c2)
```

The behaviour of agents described so far sets out how they make local decisions but they do not define how machine-level decisions are made. The Decider process is in charge of those. This centralised process collects local decisions made by the agents of a machine, offering a machine-level decision as soon as enough local decisions are gathered. This process is an abstraction that is useful for conciseness in specifying the behaviour of the machines but also for the sake of tractability. In a practical implementation of this protocol, each agent would implement the behaviour of the Decider process. Process System runs machines G and H with their respective Decider processes.

```
System = Nodes [|{|decide|}|]
(Decider(g,{},{},{}) ||| Decider(h,{},{},{}))
```

We want to ensure that the system is safe – i.e. it should stick with one decision value once a decision is made — and live — i.e. it should offer a decision event before it is allowed to deadlock. We discussed previously how we can use a type of stochastic reasoning to choose the size of the set of pickets that is necessary to achieve a given number of good and malign agents, given some parameters for our stochastic model. In our CSP model, we talk about decision sets assuming that the pickets-set size and

number of good and malign agents has been fixed, namely, the stochastic reasoning has already been used to find these numbers. So, we limit ourselves to discuss the size of decision sets that is necessary to achieve safety and liveness.

Safety may be ensured by setting a threshold that requires the participation of more than half of the benign nodes, namely, for machine m, THRESHOLD(m) ≥ GOOD(m)/2 + BAD(m) + 1, where GOOD(m)/2 is truncated integer division — we require the number of agents in each machine to be at least 2. If this threshold is set, the machine cannot decide on two different values on the same run of the protocol. Assume that agents in G supported two values, say 0 and 1, then there must be THRESHOLD(g) many agents supporting either. That implies the existence of a benign agent that has supported two values, a possibility that our protocol does not allow; a contradiction. The same reasoning holds for H's independent decision. The requirement that H must decide on G's committed values, if one exists, ensures that if they both come to a decision, their value must match. As G can only commit to one value, by the same counting argument as before, H must decide on the same value as G.

Another assumption is required to ensure liveness. We expect H to come up with a decision if G fails to do so, but the agents in H may disagree on a decision value in the case they are left to independently select it. In some actual implementations, agents may be configured to iterate if they fail to agree on a value within G or H until they eventually converge to a sufficiently agreed choice. They may achieve this in any of various way, e.g. involving coordinating input data and computing deterministically. For the sake of conciseness and tractability, we do not implement this process in the present example abstraction and we instead force enough benign agents in H to choose a common value (i.e. converge immediately), ensuring H comes to a decision. This immediate convergence is implemented by the Convergence process, which forces benign agents {0..CN} in machine H to choose the value CV - CN and CV are variables that parameterise our model. The convergent system is given by process CSystem. To achieve liveness, we need GOOD(m) ≥ THRESHOLD(m) — i.e., no malign agents are required to take part in the consensus — and that at least THRESHOLD(m)-many benign agents converge to the right value. From this inequality and the safety inequality before, one can derive the traditional lower bound on number of agents necessary for Byzantine agreement: N = 3f+1 where N is the number of agents amongst whom f are malign.

```
AlphaConvergence = {| value.h.n | n <- {0..CN-1} |}

ConvergenceAux(0) = STOP
ConvergenceAux(i) = value.h.(i-1).CV -> ConvergenceAux(i-1)
Convergence = ConvergenceAux(CN)

CSystem = System [|AlphaConvergence|] Convergence
```

Similarly to what we did for the abstract model, we use the following refinement expression to capture these properties. The specification process Spec ensures that once a decision is made, only events deciding on that value are allowed be performed and that a decision event is offered initially — it can deadlock after a decision event is performed. Process DSystem captures a projection of CSystem's behaviour onto decision events. We have used FDR to validate some instances of our model where thresholds are set in a way to ensure safety and liveness as discussed. We have also tested instances with insufficient thresholds to demonstrate how the model breaks down under those.

```
Spec = |~| m : MACHINES, v : VALUES @
      decision.m.v -> CHAOS({decision.mm.v | mm <- MACHINES})

DSystem = CSystem \ diff(Events,{|decision|})

assert Spec [F= DSystem
```

Interestingly, the inequality required to achieve safety alone does not restrict the proportion of benign agents that take part in the protocol. If we have, for instance, a single benign agent in a machine, a threshold requiring unanimity for decisions would still ensure safety. On the other hand, when both the safety and liveness inequalities are required, > 2/3 of agents should be benign. Thus, as machine G only needs to be safe, it can rely on there being as few as a single benign agent, whereas machine H, which should be safe and live, is required to have > 2/3 benign agents. Based on our stochastic calculations, for a fixed probability of an agent being malign, the number of agents that need to be selected to get a sample including at least one benign agent

should be, in general, much smaller than the number needed for a sample including > 2/3 benign agents. Therefore, the number of agents required to implement G should be, in general, much smaller than the agents required to implement H. This fact supports our claim that G should be faster at coming to a decision when compared to H, given the smaller number of agents that are required to interact.

In many cases the "pickets" making up the back-up machine H will be entire qualified population of block creators, rather than being randomly chosen. In this case the hierarchical machine will precisely be the optimistic mechanism G backed up by classic Byzantine agreement.

These formal models provide another way to consider and understand systems embodying the principles disclosed herein, but are not limiting on how embodiments might work, which may include many differing implementations. In particular, it will be appreciated by those skilled in the art that the invention has been illustrated by describing one or more specific embodiments thereof, but is not limited to these embodiments; many variations and modifications are possible, within the scope of the accompanying claims.

CLAIMS

1. A distributed computer system for performing a task, wherein the distributed computer system comprises a plurality of networked computing devices, and wherein:

a first set of the computing devices is configured to start performing a task, and to output a final result of the task after the task is completed;

a second set of the computing devices, different from the first set, is configured to determine whether the first set has output a final result of the task within a timeout period, and, if the first set has not output a final result of the task within the timeout period, to start performing the task; and

the second set of computing devices is configured, after starting performing the task and before outputting a final result of the task, to determine whether the first set has output a partial or final result since the second set started performing the task, and, if so, to output a final result that is consistent with the partial or final result output by the first set, and, if not, to output a final result of the task determined by the second set.

2. The system of claim 1, wherein the task is a decision task and the or each final result is a decision.

3. The system of claim 1 or 2, wherein the first set of computing devices is configured, before outputting a final result, to determine whether the second set of computing devices has started performing the task and, if so, to output no result or to output a final result that is the same as a final result output by the second set of computing devices.

4. The system of any preceding claim, wherein the first and second sets of computing devices are configured such that, if the first set outputs a final result, the first set does so faster than the second set outputs a final result, under some or all conditions.

5. The system of any preceding claim, wherein the first and second sets of computing devices are configured such that the second set is more likely to output a final result than the first set.

6. The system of any preceding claim, wherein the second set of computing devices is configured to be more robust than the first set of computing devices against a respective computing device of the set attempting to block the output of a final result by the respective set.

7. The system of any preceding claim, wherein the second set of computing devices is larger than the first set of computing devices.

8. The system of any preceding claim, wherein the system is a blockchain system storing a blockchain, and the computing devices are respective blockchain nodes, and wherein the task comprises determining whether to add a candidate block to the blockchain.

9. The system of any preceding claim, wherein the task is a decision task, and wherein the first set of computing devices implements a first consensus mechanisms that requires at least a first threshold number of the first set of computing devices to agree before the first set outputs a final result, and wherein the second set of computing devices implements a second consensus mechanisms that requires at least a second threshold number of the second set of computing devices to agree before the second set outputs a final result, wherein the second number is greater than the first number.

10. The system of any preceding claim, wherein the first set of computing devices is configured to output a partial result before outputting any final result.

11. The system of claim 10, wherein the task is a decision task, and wherein the partial result is a decision made by a respective computing device of the first set that is awaiting approval by the first set according to a first consensus mechanism implemented by the first set of computing devices.

12. The system of claim 11, wherein the decision task is associated with a set of decision values, and wherein the first set of computing devices are configured such that a partial result output by the first set identifies a decision value selected from the set of decision values for the task, and any final result output by the first set must confirm the decision value identified by the partial result.

13. The system of claim 12, wherein the second set of computing devices are configured such that, if a partial result is output by the first set that identifies a decision value, the final result output by the second set must confirm the decision value identified by the partial result output by the first set.

14. The system of any preceding claim, wherein the first set and second set contain at least one computing device in common.

15. The system of any preceding claim, wherein the first set of computing devices comprises one or more decider computing devices configured to implement a first consensus mechanism and to output a final result for the first set, and wherein the second set of computing devices comprises one or more decider computing devices configured to implement a second consensus mechanism and to output a final result for the second set.

16. The system of any preceding claim, wherein the sets of computing devices are configured to output a result by publishing the result such that the result is available to majority or all of the plurality of computer devices.

17. The system of any preceding claim, wherein the sets of computing devices are configured to output results comprising cryptographically signed certificates.

18. A method for performing a task in a distributed computer system comprising a plurality of networked computing devices, the method comprising:

a first set of the computing devices starting to perform the task;

a second set of the computing devices, different from the first set, determining whether the first set has output a final result within a timeout period, and, in response to determining that the first set has not output a result within the timeout period, starting to perform the task; and

the second set of computing devices, after starting performing the task and before outputting a final result of the task, determining whether the first set has output a partial or final result since the second set started performing the task, and, in response to determining that the first set has output a result since the second set started performing the task, outputting a final result that is consistent with the partial or

final result output by the first set, or in response to determining that the first set has not output a result since the second set started performing the task, outputting a final result of the task determined by the second set.

19. A computing device for use in a distributed computer system that comprises a plurality of networked computing devices, configured for performing a task, including a first set of the computing devices and a different, second set of the computing devices, wherein the computing device is configured for membership of the second set of computing devices, and is configured:

to determine whether the first set of computing devices has output a final result of the task within a timeout period, and, if the first set has not output a final result within the timeout period, to start performing the task; and

after starting performing the task and before outputting a result of the task, to determine whether the first set has output a partial or final result since the computing device started performing the task, and, if so, to output a result that is consistent with the partial or final result output by the first set, and, if not, to output a result of the task determined by the computing device.

20. Computer software comprising instructions for execution by a computing device for use in a distributed computer system that comprises a plurality of networked computing devices, configured for performing a task, including a first set of the computing devices and a different, second set of the computing devices including said computer device, wherein the instructions, which executed by said computing device, cause the computing device:

to determine whether the first set of computing devices has output a final result of the task within a timeout period, and, if the first set has not output a final result within the timeout period, to start performing the task; and

after starting performing the task and before outputting a result of the task, to determine whether the first set has output a partial or final result since the computing device started performing the task, and, if so, to output a result that is consistent with the partial or final result output by the first set, and, if not, to output a result of the task determined by the computing device.