Consensus on an Unknown Torus with Dense Byzantine Faults

Joseph Oglio, Kendric Hood, Gokarna Sharma, and Mikhail Nesterenko

Department of Computer Science, Kent State University, Kent, OH 44242, USA
{joglio@,khood5@,sharma@cs.,mikhail@cs.}@kent.edu

Abstract. We present a solution to consensus on a torus with Byzantine faults. Any solution to classic consensus that is tolerant to f Byzantine faults requires $2f + 1$ node-disjoint paths. Due to limited torus connectivity, this bound necessitates spatial separation between faults. Our solution does not require this many disjoint paths and tolerates dense faults.

Specifically, we consider the case where all faults are in one column. We address the version of consensus where only processes in fault-free columns must agree. We prove that even this weaker version is not solvable if the column may be completely faulty. We then present a solution for the case where at least one row is fault-free. The correct processes share orientation but do not know the identities of other processes or the torus dimensions. The communication is synchronous.

To achieve our solution, we build and prove correct an all-to-all broadcast algorithm $\mathcal{BAT}$ that guarantees delivery to all processes in fault-free columns. We use this algorithm to solve our weak consensus problem. Our solution, $\mathcal{CBAT}$, runs in $O(H + W)$ rounds, where H and W are torus height and width respectively. We extend our consensus solution to the fixed message size model where it runs in $O(H^3W^2)$ rounds. Our results are immediately applicable if the faults are located in a single row, rather than a column.

1 Introduction

A Byzantine process [16, 21] may arbitrarily deviate from the prescribed algorithm. This is the strongest fault that can affect a process in a distributed system. The fault is powerful enough to straddle the realm of fault tolerance and security as it may model either a device failure or a malicious intruder.

In the presence of Byzantine faults, the common task for correct processes is to come to an agreement or consensus. The power of the faults may be abridged with cryptography [1, 9, 14] or randomization [4, 10, 24]. If neither primitive is available, the solutions require that the number of correct processes is large enough to overwhelm the faulty ones.

If the topology of a network is considered, the consensus problem is further complicated as faulty processes, even small in absolute number may isolate some correct processes and prevent them from achieving consensus. In a general topology, it is known that consensus is solvable only if the network is at least

$2f + 1$-connected [8, 11, 16]. However, such connectivity demands a dense network with large node degree which limits the scalability of a solution achieved this way.

In this paper, we study Byzantine-robust consensus on a torus. Its fixed degree and small diameter makes torus an attractive architecture for distributed computing and storage tasks [3, 12]. Torus connectivity is 4. Hence, according to classic connectivity bounds, it may not tolerate more than 1 fault. To increase tolerance, the fault locations must be restricted. One approach is to make the faults sparse. That is, the faulty processes need to be positioned far enough apart so that $2f+1$ connectivity is preserved [17]. Such a solution fails if the faults are located close to each other. In this paper, we address dense Byzantine faults on a torus of unknown dimensions. However, to achieve tolerance, we restrict fault location to a single column.

Related work. There are a number of solutions optimizing consensus in incomplete topologies [2, 7, 18, 25, 26]. Chlebus et al. [7] optimize the speed of achieving Byzantine consensus in arbitrary topologies. The connectivity is subject to $2f+1$ bound. Alchieri et al. [2] study synchronous Byzantine consensus with unknown participants. They use participant detectors to establish network membership. Tseng and Vaidya [25] explore consensus in directed graphs. Winkler et al. [26] study consensus in directed dynamic networks. Oglio et al. [19] solve Byzantine consensus in Euclidean space.

Potentially, network topology may be discovered despite Byzantine faults. This simplifies consensus solution. Nesterenko and Tixeuil present two generic Byzantine-tolerant topology discovery algorithms [18]. However, neither algorithm is suitable for our task: their first algorithm raises an alarm once an inconsistency is discovered without completing the task. Their second algorithm requires $2f + 1$ connectivity.

Let us discuss consensus on toruses and related topologies. Several papers [5, 15, 22] consider the problem of Byzantine-tolerant reliable broadcast on an infinite grid or a torus in radio networks. In such a network, all processes within a particular distance from the sender receive the message simultaneously. Due to this limitation, their results are not immediately applicable to our model.

Kandlur and Shin [13] consider a synchronous Byzantine-tolerant broadcast on a torus. In their approach, each message is delivered over a fixed number of node-disjoint paths. The correct message is recovered so long as the majority of paths bypass faulty nodes. Since torus connectivity is 4, this approach may tolerate at most one fault. Maurer and Tixeuil [17] study a Byzantine-tolerant broadcast on a torus. In their model, the byzantine torus dimensions are not known. Their solution assumes that the faulty nodes are sparse, i.e. they are located far enough apart such that there is sufficiently many node disjoint paths between the sender and any of the receivers to ensure that the influence of the faulty nodes is countered.

Thus, to the best of our knowledge, previous work has not addressed dense Byzantine faults on a torus.

Paper contribution and approach. We consider the synchronous model, bidirectional communication, no cryptographic primitives or randomization. The topology we study is a torus whose dimensions: height H and width W are unknown to the processes. All processes share orientation. We assume that all faults are located in a single column.

We consider the weaker consensus where only processes in fault-free columns must agree on a value. We prove that even this version is not solvable if a single column is completely faulty.

We examine the case where at least one row is fault-free. To counter faulty column influence, we assume that $W \geq 5$. To solve this weak consensus problem, we first present an all-to-all broadcast algorithm $\mathcal{BAT}$ that guarantees delivery to fault-free columns. We prove it correct and show that it runs in $O(H + W)$ rounds in the LOCAL model [21] where messages are of arbitrary size. We then use $\mathcal{BAT}$ to build the consensus algorithm $\mathcal{CBAT}$. We prove it correct and show that it runs in $O(H + W)$ rounds in LOCAL.

We then extend $\mathcal{BAT}$ and $\mathcal{CBAT}$ to use fixed-size messages and estimate their running time in the CONGEST model [23]. We determine that the fixed-size message $\mathcal{BAT}$ runs in $O(H^2W)$ rounds and $\mathcal{CBAT}$ runs in $O(H^3W^2)$ rounds. Our results are immediately applicable to the case of faults located in a single row rather than column.

Let us now introduce our solution approach. To counter the faults, we leverage the processes' shared knowledge of the network topology. The data is first propagated along the column and the influence of possibly densely packed column of faults is neutralized. The single correct process is guaranteed to relay data across the faulty column. This horizontally propagated data is then disseminated through the fault-free columns to reach all correct processes there.

Paper organization. In Section 2, we introduce our notation, state the problem and prove the necessity of a fault-free row. In Section 3, we present and prove correct our all-to-all broadcast algorithm $\mathcal{BAT}$. In Section 4, we use $\mathcal{BAT}$ to construct and prove correct our consensus algorithm $\mathcal{CBAT}$. In Section 5, we describe how the algorithms can be modified for fixed message size and estimate their run time. We conclude the paper by describing further research directions in Section 6.

2 Notation, Problem Statement, Fault Constraints

Computation model. A process p contains variables and actions. We denote variable *var* of process p as *var.p*. If process p maintains variable *var* about process q, we denote it as *var.q.p*. A rectangular grid graph of processes is a Cartesian product of two chain graphs. Such a grid graph is embedded on a plane as a matrix of rows and columns. A *torus grid graph*, or *torus* for short, is formed by starting with a grid graph and connecting corresponding leftmost/rightmost and top/bottom processes with edges.

Every process in the torus has a unique identifier. An adjacent process is a *neighbor*. In a torus, every process has exactly four neighbors. Each process

knows the identifiers of its neighbors: *left*, *right*, *up* and *down*. All processes share the orientation. That is, for any two processes p and q if $up.p = q$ then $down.q = p$ and if $left.p = q$ then $right.q = p$. We refer to the shared orientation as North, East, West and South. The torus dimensions are unknown to the processes. That is, they do not know the height H or the width W of the torus.

The system is completely synchronous. The operation proceeds in rounds. In every round, each process receives all pending messages sent to it, does local calculations and sends messages to its neighbors to be received in the next round. In one round, a process may thus receive multiple messages from the same or from different neighbors and then send multiple messages to neighbors. A *computation* is an infinite sequence of such rounds.

For most of the paper, we assume that size of the message is arbitrary. We lift this assumption later in the paper.

Process faults. Processes are either correct or faulty. A *correct* process follows the algorithm while a *faulty* process behaves arbitrarily. A faulty process is always *black*. A correct process is *grey* if it has at most one black process in its column, it is *white* otherwise. The colors of individual processes are applied to the rows and columns of the torus: grey-white, black-white, etc. Refer to Figure 1 for torus depiction and fault location.

Broadcast. Consider the problem where each process p is input an arbitrary initial value $initVal.p$, and p must share this value with every other process q so that $val.q.p = initVal.p$

Definition 1. *In the* Weak Synchronous All-to-All Broadcast Problem *every white process p must stop and for each white process q, p must output $val.q.p$ such that $val.q.p = initVal.q$.*

Consensus. In *Binary Consensus*, the input value $initVal.p$ is restricted to either 0 or 1. Each process must output an irrevocable decision v with the following three properties:

- *agreement* – no two correct processes decide differently;
- *validity* – if there are no faults and for every process p, $initVal.p = v$, then p decides v;
- *termination* – every correct process eventually decides.

Definition 2. *In* strong consensus*, the above properties apply to every grey and white process, i.e. to each correct process. In* weak consensus*, the properties apply only to white processes.*

Impossibility. Let us outline the area of the possible. Strong consensus requires $2f + 1$ connectivity [8]. The connectivity of torus is 4, so there is no algorithm that solves strong consensus on a torus with $f > 1$ and arbitrary fault location.

If faults are restricted to a single column but may occupy the complete column, consensus is still impossible. This holds true even if the processes know the torus dimensions. Intuitively, even if correct processes are connected, they may not be able to distinguish their faulty and non-faulty neighbors and agree on a value. The below theorem formalizes this observation.

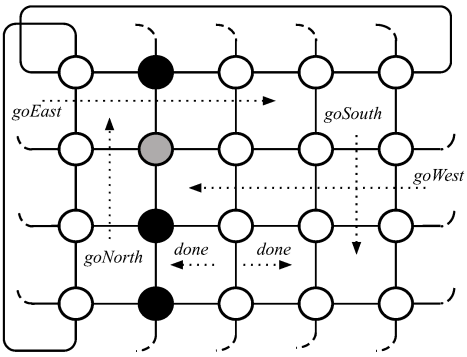

Fig. 1. Torus orientation, fault location, message types and message propagation direction in $\mathcal{BAT}$.

Theorem 1. *There is no algorithm that solves consensus, weak or strong, on a torus with a faulty column.*

Proof. Assume the opposite: there is such an algorithm $\mathcal{A}$ hat solves weak consensus on a torus with a completely faulty column. Since there are no grey processes, the requirements of weak and strong consensuses are identical. Consider a torus T of height 1 and some width W. Since the algorithm $\mathcal{A}$ is a solution the consensus problem on a torus, it should be able to solve it on T. However, this topology is a ring. Its connectivity is 2. According to Dolev et al. [8], the consensus is solvable only if the connectivity of the network is at least $2f+1 = 3$. That is, contrary to our initial assumption, $\mathcal{A}$ may not solve consensus on T. Hence the theorem. □

3 $\mathcal{BAT}$: Byzantine-Tolerant Broadcast to All-to-All on a Torus

Overview. We present algorithm, we call $\mathcal{BAT}$, that solves the Weak Synchronous All-to-All Broadcast Problem. The code for $\mathcal{BAT}$ is shown in Algorithm 1. The functions used in $\mathcal{BAT}$ are in Algorithm 2. Our notation is loosely based on UNITY programming language [6]. Each process starts $\mathcal{BAT}$ by sending initial messages in Line 14. The execution of the rest of the actions are *receive actions*. Such an action is guarded by the corresponding message receipt. It is executed only when this message is sent to the receive process. The actions of the algorithm are grouped into phases: North, East-West, South and Decision. The algorithm is designed such that white processes synchronize their transitions through these phases.

Once done, each process sends the collected column data to its *left* and *right* neighbors in the East-West Phase. The data is sent in both directions for verification to counteract the actions of the grey and black processes in the row.

Algorithm 1: $\mathcal{BAT}$: Byzantine All-to-All broadcast on a Torus

```
   Input: initVal
 1 Constants:
 2 p                                                          // process identifier
 3 up, down, left, right                                      // neighbor identifiers
 4 Variables:
 5 northDone, initially false
 6 column, initially ⟨⟩              // seq. of value-id pairs received from down neighbor
 7 rowLeft, rowRight, initially ⟨⟩   // columns from resp. left and right neighbors
 8 matrix, initially ⟨⟩                                       // results matrix to output
 9 eastWestDone, initially false          // one of horizontal neighbors decided
10 Phases:
11 North:
12     Initial action:
13     add ⟨initVal, p⟩ to column                             // initiate North Phase
14     send goNorth(initVal, p) to up
15     Receive action:
16     receive goNorth(v, id) from down ⟶
17         if not northDone then                  // has not started East-West Phase
18             if id ≠ p then
19                 add ⟨v, id⟩ to column
20                 send goNorth(v, id) to up
21             else                                // p receives its own value back
22                 northDone ← true
23                 send goEast(column, left, p, right) to right
24                 send goWest(column, left, p, right) to left
25 East-West, Receive actions:
26     receive goEast(c, l, id, r) from left ⟶
27         if id ≠ p then
28             add ⟨c, l, id, r⟩ to head of rowLeft
29             send goEast(c, l, id, r) to right
30         else
31             m ← match(⟨column, left, p, right⟩ + rowLeft,
32                       ⟨column, left, p, right⟩ + rowRight)
33             if matrix = ⟨⟩ and m ≠ ⟨⟩ then
34                 matrix ← m
35                 Output: matrix
36                 send goSouth(matrix, p) to down          // start South Phase
37                 send done to right and left              // start Decision Phase
38                 if eastWestDone then stop
39     receive goWest(c, l, id, r) from right ⟶
           // handle similar to goEast, add ⟨c, l, id, r⟩ to tail of rowRight
40 South, Receive action:
41     receive goSouth(m, id) from up ⟶
42         if id ≠ p then
43             send goSouth(m, id) to down
44             if matrix = ⟨⟩ then          // South Phase did not reach p yet
45                 matrix ← m
46                 Output: matrix
47                 send done to right and left          // start Decision Phase
48                 if eastWestDone then stop
49 Decision, Receive action:
50     receive done from direction ⟶
51         if not eastWestDone then
52             eastWestDone ← true
53             if matrix ≠ ⟨⟩ then stop
```

Algorithm 2: $\mathcal{BAT}$ functions

```
54 Functions:
55 match(rleft, rright):
56     cleft ← consistent(rleft)
57     cright ← consistent(rright)
58     if cleft ≠ ⟨⟩ and cright ≠ ⟨⟩ then
           // cleft = cright
59         if |cleft| = |cright| and
60         (∀i : 1 ≤ i < |cleft| :
61           s(i) ≡ ⟨lci, lli, lidi, lri⟩ ∈ cleft
62           s(i) ≡ ⟨rci, rli, ridi, rri⟩ ∈ cright :
63           lci = rci and lli = rli and lidi = ridi and lri = rri) then
               // return columns of cleft
64             return ⟨∀i : 1 < i ≤ |cleft| : s(i) ≡ ⟨ci, li, idi, ri⟩ ∈ cleft : ci⟩
65     return ⟨⟩

66 consistent(clmn):
67     if s ≡ ⟨·, ·, id, ·⟩ is unique in clmn and
68       exists at most one i : 1 ≤ i < |clmn| : cminus = clmn \ s(i) and
69       exists at most one j : 1 ≤ j ≤ |clmn| :
70             cplus = cminus insert ⟨⊥, l, id, r⟩ at position j in cminus
71       ∀i : 1 ≤ i < |cplus|, j = (i + 1 mod |cplus|) :
72             s(i) ≡ ⟨ci, li, idi, ri⟩ ∈ cplus
73             s(j) ≡ ⟨cj, lj, idj, rj⟩ ∈ cplus :
74             idi = lj and ri = idj then
75       return cplus
76     else
77       return ⟨⟩
```

If the data received from both directions match, each process starts the South Phase where the confirmed data is sent to *down* neighbor. This data is a matrix of values from all processes in the torus.

Due to the actions of the black processes in the black-white rows, the white processes may receive corrupted values and fail to complete the South Phase on time. Once the data starts propagating down the column in the South Phase, the white processes synchronize, receive correct matrix and output it. After output, processes enter the Decision Phase. This phase ensures termination of white processes.

$\mathcal{BAT}$ details. The input for each process in the algorithm is an arbitrary value *initVal*. Each process p in $\mathcal{BAT}$ has its own identifier as well as the ids of its up, down, left and right neighbors.

Each process p maintains the following variables. The process has *column*, where it gathers pairs $\langle v, id \rangle$ of values and identifiers of its column in the North Phase. Boolean *doneNorth* signifies whethe p completed its North Phase. After it is done, the column values are exchanged across the row in two directions: left and right in the East-West Phase. These column values are collected in *rowLeft* and *rowRight*. Once the values are matched across the row, they are propagated through the column in the South Phase and collected in *matrix*. Variable *doneNeighbor* records the neighbor identifiers that made the decision to ensure proper termination.

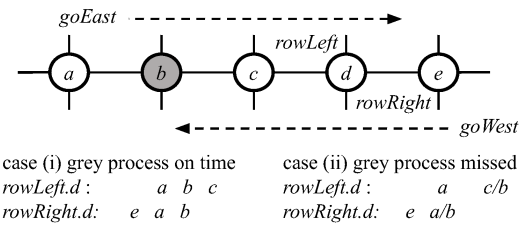

Fig. 2. Matrix accumulation during the East-West Phase of $\mathcal{BAT}$. The matrix is accumulated at process d as it collects messages *goEast* and *goWest* from its left and right neighbors respectively. In case (i), the grey process b starts East West Phase together with white processes. In case (ii), b starts it one round earlier.

Let us discuss $\mathcal{BAT}$ operation during each phase in detail. Refer to Figure 1 for the messages that processes send and their propagation directions. All non-black processes simultaneously initiate the North Phase by sending their input values to the *up* neighbors in a *goNorth* message (see Line 15). Then, each process p starts collecting the values of its column processes by receiving *goNorth* from the *down* neighbor and propagating it upward. This continues until p receives its own message back. Once received, p initiates the East-West Phase (Line 23).

For that, p sends the collected column to its *left* and *right* neighbor in messages *goWest* and *goEast*, respectively. Together with the column, p sends its own identifier and the identifiers of *left* and *right*. This allows the recipients to reconstruct the sequence of the processes in the row if one of the entries is missing. Flag *northDone* ensures that each process sends *goWest* and *goEast* at most once.

Due to the actions of the black processes in its column, even if some grey process completes its own North Phase, it may do so out of sync with the White processes. However, the receipt of *goEast* or *goWest* from its neighbor, enables g's corresponding receive action of the East-West phase. Thus, even though g itself is out of sync, it relays these messages to the white processes in its row allowing them to proceed with the execution the East-West Phase.

The East-West Phase is the most complicated part of the algorithm. Let us illustrate its operation with an example. Please see Figure 2. Assume a torus row contains processes a, b, c, d and e. All of them are white, except for b which is grey. Each white process completes the North Phase in the same round and starts the East-West Phase by sending the collected column to its right neighbor in a *goEast* message and to its left neighbor in *goWest* message. The grey process may (i) complete its North Phase together with the white processes or (ii) miss it and either complete it in a different round or not at all.

Let us first describe case (i). Once a process receives either *goEast* or *goWest*, the process records what the message carries and sends it further in the same direction. Thus, in the first round of the East-West Phase, d receives messages from c and e, in the second b and a and so on. As *goEast* message arrives to d from its left neighbor, d inserts its contents to the head of *rowLeft.d*.

When *goWest* message arrives to d from the right, d adds its contents to the tail of *rowRight.d*. Figure 2 shows the contents *rowLeft.d* and *rowRight.d* after three rounds of the East-West Phase when d received messages from c, b, and a from the left and e, a, and b from the right. The East-West Phase proceeds until the process receives a message from itself. In the example in Figure 2, this happens in two more rounds, after d receives messages from e then d from the left and c then, possibly, d from the right.

In case (ii), the grey process b completes the North Phase in a round different from the white processes. In Figure 2, b completes its North Phase one round earlier. Therefore, this message reaches d together with a message from a from the right, and together with message from c from the left.

After the process receives the message from itself, either from the right or the left, it compares the contents of *rowLeft* and *rowRight* by invoking function **match()** (Line 32). The operation of this function is somewhat complex since grey process may be out of synchrony with the rest of its row. If the contents of the two variables do not match, Function **match()** returns the data stripped of a mismatched column of the grey process. The resultant list of columns: *matrix* is the output of $\mathcal{BAT}$. This *matrix* is sent to the *down* process in a *goSouth* message. This begins the South Phase (Line 36).

$\mathcal{BAT}$ operation is such that all white processes either initiate the South Phase in the same round or do not initiate the South Phase at all. Indeed, due to the actions of the black processes in the black-white rows, the white processes may not get the matching data and fail to start their own South Phase. However, the white processes in the grey-white row are guaranteed to start the South Phase. The South Phase started by these processes, re-synchronizes white processes and propagates the correct *matrix* information in *goSouth* message. Once a process receives the missing information, it outputs it (Line 44).

Yet, the processes should not terminate. Indeed, black processes may force a grey process in the black-grey column to receive a *goSouth* message at any time. The grey processes may output the result and consider its mission accomplished. If a grey process terminates, its halting prevents it from forwarding messages from white processes in the East-West Phase.

To ensure proper termination, processes execute the Decision Phase. After outputting the decision matrix, each process sends *done* message to its horizontal, i.e. left and right, immediate neighbors. The process stops if it receives at least one of them (Line 52) and it has the decision matrix. One of the horizontal neighbors is guaranteed to be white. Thus, if a process obtains the decision matrix and gets one *done*, then its white neighbors may not be in the middle of the East-West Phase and this process may safely stop.

Discussion on $\mathcal{BAT}$ functions. Let us now discuss the functions used in $\mathcal{BAT}$. They are shown in Algorithm 2. Function **match()** accepts the input received from the West – *left*, and East – *right* during the East-West Phase. The entries of *left* and *right* are of the following format: the column c received by the original

sender, its left neighbor identifier l, its own identifier id and its right neighbor id r.

First, *left* and *right* are individually checked for internal consistency in function **consistent()**. As process p receives data from either East or West, the left and right neighbors of the subsequent entries should match. In grey-white row, this is true for all entries except for possibly the grey process that may start its East-West phase earlier or later than the white ones. In this case, the grey process entry may arrive out of order. That is, it may arrive together with another white process entry while its own spot in the sequence remains empty.

Function **consistent()** finds a potential single entry that is out of sequence as well as a single process gap (Line 69). Specifically, **consistent**() checks whether there exists at most one element in its parameter *clmn* such that if it is removed, *cminus* is produced, and then another element is possibly added to *cminus* producing *cplus*. This added entry contains $\perp$ for the column values and the l, id, r tuple are such that the left and right neighbors in the consequent entries in resultant *cplus* match. In this case **consistent**() returns *cplus*. Consider case (ii) shown in Figure 2. Observe what *rowLeft* and parameter *left* contain:

$$(1)\langle\cdot, c, d, e\rangle, (2)\langle\cdot, d, e, a\rangle, (3)\langle\cdot, e, a, b\rangle, (4)\perp, (5)\langle\cdot, b, c, d\rangle/\langle\cdot, a, b, c\rangle$$

That is, the third round entry is empty and the fifth round entry contains the data originated by process c and the grey process b. Process b sends its data out of sync with the white processes. Function **consistent**(), on the basis of the adjacent entries, reconstructs the fourth entry and drops the extra fifth entry sent by b:

$$(1)\langle\cdot, c, d, e\rangle, (2)\langle\cdot, d, e, a\rangle, (3)\langle\cdot, e, a, b\rangle, (4)\langle\perp, a, b, c\rangle, (5)\langle\cdot, b, c, d\rangle$$

The corrected entry is stored in *cleft* by **match()**. Similar manipulation by **consistent**() yields *cright*. Function **match**() determines that *cleft* is equal to *cright*. That is, the process p gets the same data from both East and West. In this case **match**() returns the matrix of columns stored by *cleft* and *cright*.

Theorem 2. *Algorithm $\mathcal{BAT}$ solves the* Weak Synchronous All-to-All Broadcast Problem *on an unknown torus with Byzantine faults in at most one column with at least one correct row in at most $2H + 2 + W$ rounds.*

The above theorem states the correctness and efficiency of $\mathcal{BAT}$. See [20] for proof details. Let us discuss the intuition for the correctness proof. It is relatively straightforward to show that all white processes complete the first, North, Phase in $H + 1$ round and collect the true contents of their respective columns. By induction on the number of rounds, we show that each process p collects a column of true data about any process q in the same row, so long as the processes between p and q, in the direction of information propagation are non-black. Indeed, the non-black processes do not impede data propagation.

We then separately consider white processes in (a) grey-white rows and (b) black-white rows. In the East-West Phase, the information propagates in two

directions. We show that during this phase, in a grey-white row, in round $H + 1 + W$, the information spreading in two directions matches. The match is up to the potentially misplaced data from the single grey process. We prove that this information is true to the original white process data throughout the matrix. In the black-white row, we show that a white process either completes the phase in $H + 1 + W$ round with correct white process data or not at all.

That is, we show that each white process, regardless of the row location, that complete the East-West Phase, holds correct information about all the white processes in the matrix. Moreover, each white process in the grey-white row is guaranteed to complete this phase.

We then consider the South Phase. There, we show that it is completed in an additional H rounds, and all the white processes are updated with the correct matrix information before termination. This completes the proof of Theorem 2.

The computation model that we consider $\mathcal{BAT}$ is called LOCAL. It assumes unlimited size messages. Theorem 2 shows that $\mathcal{BAT}$ completes in $O(H + W)$ rounds in LOCAL.

4 $\mathcal{CBAT}$: Consensus Using $\mathcal{BAT}$

Description. Observe that $\mathcal{BAT}$ cannot immediately be used to solve consensus since, after its completion, the outputs of each process differ by the values of black and grey processes. An individual white process is not able to determine the color of the senders. Therefore, if the white process makes the consensus decision on the basis of the single execution of $\mathcal{BAT}$, the Byzantine senders may cause white processes to disagree on their outputs.

Instead we use $\mathcal{BAT}$ to select a leader process and agree on the input value of this leader. A particular difficulty is presented if the selected leader process is faulty. In this case, the leader may send different values to different processes. To prevent that, the processes use $\mathcal{BAT}$ again to exchange received values, determine whether the leader is consistent in its transmission of its initial value, and, if not, replace the leader . This determination has to proceed despite the inconsistent information provided by the faulty processes in this second exchange.

Let us describe the algorithm $\mathcal{CBAT}$ that achieves consensus. It is shown in Algorithm 3. We assume that the number of columns $W \geq 5$. The algorithm has three sequentially executing stages: Broadcast, Confirm and Decide. In the first two stages $\mathcal{CBAT}$ executes $\mathcal{BAT}$.

In the Broadcast Stage, the processes exchange the initial input values. In the Confirm Stage, they transmit the complete matrix they received during Broadcast. At the end of Confirm, each white process receives $H \times W$ matrices of such values. In the Decide Stage, every white process independently arrives at the consensus decision.

Let us describe how this decision is computed. Each process p deterministically selects a leader process ldr. In the algorithm, it is the process with the highest identifier. Then, out $H \times W$ matrices that process p receives in the second stage, p composes the matrix M_L of values sent to all other processes

Algorithm 3: $\mathcal{CBAT}$

```
   Input: v boolean                                              // consensus input value
78 Variables:
79 M_B ≡ {v_ij : 1 ≤ i ≤ H, 1 ≤ j ≤ W}        // matrix of votes received by each process
80 M_C ≡ {M_ij : 1 ≤ i ≤ H, 1 ≤ j ≤ W}      // votes reported as received by each process
81 M_B and M_C are initially ⊥
82 Stages:
83 Broadcast:
84    M_B ← BAT(v)
85 Confirm:
86    M_C ← BAT(M_B)
87 Decide:
      // deterministically select leader
88    ldr ← highestID(M_B)
89    let v_mn ∈ M_B be the input value of ldr
90    let C_L ∈ M_B                                          // leader's column in M_B
91    let M_L = {v_ij : v_mn ∈ M_ij ∈ M_C}     // v sent by ldr, received by every process
92
      // two columns with ⊥ elements or
      // two columns with 1/0 elements or
      // two columns whose values differ from rest
93    if ∃ q ≠ r and ∃ e, f :
94       v_qe, v_rf ∈ M_L \ C_L :
95       v_qe = v_rf = ⊥
96       or
97       ∃ w ≠ x, y ≠ z and ∃ a, b, c, d :
98       v_wa, v_xb, v_yc, v_zd ∈ M_L \ C_L :
99       v_wa = v_xb = 0 and v_yc = v_zd = 1 then
            // select a new leader from a different column
100         ldr ← highestID(M_B \ C_L)
            // recompute M_L and C_L
101   Output: majority(M_L \ C_L)
```

by the leader as these processes report to p. Process p examines these values for inconsistencies. The inconsistencies are as follows: either the values differ in more than one column; or the values contain $\perp$, a sign that $\mathcal{BAT}$ detected faulty values, in more than one column; or the values in at least two columns differ from the rest. In the latter case, since the total number of columns is at least 5, and the leader column is not considered, there are at least two columns that are different from at least two other columns. In all cases, these inconsistencies can be determined by all correct processes. Once the faulty process is detected, the whole column where all the faulty processes may be located is also discovered. Therefore, p selects a new leader from a different column. Once each process selects the leader, that leader may still be faulty and send arbitrary values to other processes. However, these values are consistently stored in the M_L matrix. Each process decides on the majority of values in this matrix.

Correctness proof. Note that the below discussion relies on correctness of $\mathcal{BAT}$ proved in Theorem 2. Denote $val.a$ the input value $initVal$ of some fixed process that is reported by process a after the Broadcast Stage of $\mathcal{CBAT}$. Denote $val.a.b$ the same fixed process value received by process b from process a after the Confirm Stage.

Lemma 1. *Let v, w, x be white processes executing $\mathcal{CBAT}$ and distributing the value input to some process u. Then, after the Confirm Stage of $\mathcal{CBAT}$, if u is white then $val.v.x = val.w.x$; and $val.v.x = val.v.y$ regardless of the color of u.*

Proof. Let us address the first claim of the lemma where u is white. In the Broadcast Stage, processes are sending their individual input values using $\mathcal{BAT}$. This includes $initVal$ of process u. If the sender u is white and the recipients v and w are white, then, according to Theorem 2, $initVal = val.v = val.w$. During the Confirm Stage, each process, including v and w, sends the complete matrix of received values, including $val.v$ and $val.w$ to all processes, including x using $\mathcal{BAT}$. Since v, w, and x are white, according to Theorem 2, $val.v = val.v.x$ and $val.w = val.w.x$. That is, $val.v.x = val.w.x$. This proves the first claim of the lemma.

Let us consider the second claim that should hold regardless of the color of u. Consider the value $val.v$ stored at v after the Broadcast Stage. During Confirm, this value is sent to all processes including x and y. If v, x and y are white, according to Theorem 2 $val.v = val.v.x = val.v.y$. Hence the second claim of the lemma also holds. □

Lemma 2. *Every white process selects the same leader in the Decide Stage of $\mathcal{CBAT}$.*

Proof. For each white process p, algorithm $\mathcal{BAT}$ is guaranteed to produce a matrix that has the same identifiers, configuration and size. This means that all processes select the same ldr in Line 88 of $\mathcal{CBAT}$.

Let us consider whether each process changes its leader in Line 100. This happens if process p detects inconsistencies in at least two columns excluding the leader column. Process ldr is either correct or faulty.

If ldr is correct, then according to the first claim of Lemma 1, the entries for ldr in M_L at p are the same, except for possibly a single column of faulty processes. This means that if ldr is correct, none of the white processes changes its leader after initial selection.

If ldr is faulty, then according to the second claim of Lemma 1, the corresponding entries for ldr in M_L for all white processes are equal. That is, if processes p and q are white, then if entry $m_{ij} \in M_L$ in p is equal to $m_{ij} \in M_L$ in q.

That is, if two non-leader columns are inconsistent, then all white processes detect that. That is, if one white process changes leader, all white processes change leaders also. If the processes change leaders, by the operation of the algorithm, they select the same leader. Hence the lemma. □

Lemma 3. *All white processes in $\mathcal{CBAT}$ output the same value. If the selected leader ldr is correct, they input $initVal.ldr$.*

Proof. Every white process p outputs the majority of values in $M_L \setminus C_L$. Due to Lemma 2, white processes in $\mathcal{CBAT}$ select the same leader. The leader may be either correct or faulty.

If the leader is correct, then, According to Lemma 1, all white entries in $M_L \setminus C_L$ are equal to *initVal.ldr*. Only one column in $M_L \setminus C_L$ may be non-white with potentially arbitrary values. Since we assume that the torus contains at least 5 columns, the majority of $M_L \setminus C_L$ is equal to *initVal.ldr* and that is what the white process p outputs.

Let us consider the case of faulty leader. White processes change the leader only when they detect that the original leader is faulty. They select it from a different column, therefore, the new leader is correct and the previous reasoning to the output value applies.

The remaining case is that of a faulty leader and no leader change. According to the operation of the algorithm, after leader selection at most one column in $M_L \setminus C_L$ has inconsistencies. The rest of the entries hold the same value $x \in 0, 1$. According to the second claim of Lemma 1, the inconsistencies are in the same column of $M_L \setminus C_L$ for each white process. This means that the rest of the values in the matrix are x. Since white processes output the majority value and the number of columns is at least 5, all white processes output the same value x. □

Theorem 3. *$\mathcal{CBAT}$ solves weak consensus on an unknown torus whose width is at least* 5 *with Byzantine faults in at most one column and with at least one correct row.*

Proof. We prove the theorem by showing that $\mathcal{CBAT}$ satisfies all three properties of consensus. The agreement property of the consensus requires that all white processes output the same value. Lemma 3 indicates that $\mathcal{CBAT}$ satisfies that property. The validity property states that in case there are no faults and all processes are input the same value, they should all output the same value. According to Lemma 3, if the leader is correct, all correct processes output its input value. Hence, if all processes are correct, and they are all input the same value, they select one process as the leader and output this value. Therefore, $\mathcal{CBAT}$ satisfies validity.

Let us address termination. The consensus requires that each correct process terminates. $\mathcal{CBAT}$ sequentially executes three stages. The first two stages are executions of $\mathcal{BAT}$ and the last one is a finite computation. According to Theorem 2, $\mathcal{BAT}$ terminates. Hence, all three stages of $\mathcal{CBAT}$ terminate and this algorithm satisfies the termination property of the consensus. □

Algorithm $\mathcal{CBAT}$ sequentially executes $\mathcal{BAT}$. Each $\mathcal{BAT}$ completes in $O(H+W)$ rounds in LOCAL. This means that $\mathcal{CBAT}$ also completes in $O(H + W)$ rounds.

5 Extension to Fixed Message Size

As presented, $\mathcal{BAT}$ is assumed to operate with unlimited size messages. However, it can be modified to operate with fixed size messages as follows. Observe that the messages get progressively larger as they accumulate the data about the torus. In the first phase, the North Phase, the messages are fixed size since each process only transmits its identifier and its input value. At most one message is

sent per link per round. After the completion of this phase, the white processes discover the height H of the torus. In the next phase, the East-West Phase, the processes exchange messages whose size is proportional to the height of the torus. Since white processes know the torus size, this message may be replaced by H fixed size messages transmitted over H rounds. Due to the operation of the algorithm, at most 2 messages are transmitted per round in the East-West Phase. In the fixed size implementation, each process waits for $2H$ rounds to receive appropriate messages. The black processes may deceive the grey process and assume the larger torus height. This would make the grey process transmit a message larger than H rounds during the East-West Phase. This, in turn, may prevent the correct message from being transmitted in the same round. To eliminate that, the blocks of the two messages need to be transmitted in the round-robin manner. The South Phase message transmits the complete matrix. So the fixed size implementation has to wait for $H \cdot W$ rounds to receive a single matrix.

The fixed message size model is called CONGEST. Let us estimate the running time of the modified algorithm in CONGEST. The four phases of the original $\mathcal{BAT}$ take $O(H)$, $O(W)$, $O(H)$ and $O(1)$ rounds, respectively. The above argument shows that in each respective phase, the modified $\mathcal{BAT}$ needs to send $O(1)$, $O(H)$, $O(HW)$ and $O(1)$ sequential fixed-size messages per round. Hence, the number of rounds in fixed-size message $\mathcal{BAT}$ is dominated by the third phase and is in $O(H^2W)$.

Let us now discuss the fixed message size modification of $\mathcal{CBAT}$. In the first stage, $\mathcal{CBAT}$ uses $\mathcal{BAT}$ to broadcast constant-size decision values. Hence, its run time is in $O(H^2W)$. In the second stage, each process sends a complete $H \cdot W$ matrix. Hence, the run time of this stage, and of the whole algorithm is in $O(H^3W^2)$.

6 Conclusion and Future Work

Algorithm $\mathcal{BAT}$ assumes that all the faults are in the same column. In future research, the following conjectures are worth investigating. We believe that solving the problem with faulty processes spread across multiple columns, one fault per row, is possible but requires substantial modification of the algorithm. We suspect that solving the problem for the case of more than one fault per row is not possible.

To achieve the solution presented in this paper, we assumed shared process orientation. It is interesting to explore how important this assumption is to the solution. That is, whether it is possible to solve the problem without shared orientation.

In this paper we presented a Byzantine-tolerant consensus algorithm that exploits the knowledge of the network type – torus, to exceed the tolerance bound presented by the general consensus algorithm. Our study then opens the following question: what network types have similar properties? To put another way, what specifically makes torus Byzantine-fault resistant and can this property be generalized? We believe that this is a fruitful avenue of future research.

References

1. I. Abraham, S. Devadas, K. Nayak, and L. Ren. Brief announcement: Practical synchronous byzantine consensus. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
2. E. A. Alchieri, A. N. Bessani, J. d. Silva Fraga, and F. Greve. Byzantine consensus with unknown participants. In *International Conference On Principles Of Distributed Systems*, pages 22–40. Springer, 2008.
3. P. W. Beame and H. L. Bodlaender. Distributed computing on transitive networks: the torus. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 294–303. Springer, 1989.
4. M. Ben-Or. Another advantage of free choice (extended abstract) completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.
5. V. Bhandari and N. H. Vaidya. On reliable broadcast in a radio network. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 138–147, 2005.
6. K. Chandy and J. Misra. *Parallel program design: A foundation*. Addison Wesley Publishing Co., 1988.
7. B. S. Chlebus, D. R. Kowalski, and J. Olkowski. Fast agreement in networks with byzantine nodes. In *34th International Symposium on Distributed Computing (DISC 2020)*, 2020.
8. D. Dolev. The byzantine generals strike again. *Journal of algorithms*, 3(1):14–30, 1982.
9. D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
10. P. Feldman and S. Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.
11. M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.
12. P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, pages 19–24, 2004.
13. D. D. Kandlur and K. G. Shin. Reliable broadcast algorithms for harts. *ACM Transactions on Computer Systems (TOCS)*, 9(4):374–398, 1991.
14. J. Katz and C.-Y. Koo. On expected constant-round protocols for byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009.
15. C.-Y. Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 275–282, 2004.
16. L. LAMPORT, R. SHOSTAK, and M. PEASE. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
17. A. Maurer and S. Tixeuil. Byzantine broadcast with fixed disjoint paths. *Journal of Parallel and Distributed Computing*, 74(11):3153–3160, 2014.
18. M. Nesterenko and S. Tixeuil. Discovering network topology in the presence of byzantine faults. In *International Colloquium on Structural Information and Communication Complexity*, pages 212–226. Springer, 2006.
19. J. Oglio, K. Hood, G. Sharma, and M. Nesterenko. Byzantine geoconsensus. In *International Conference on Networked Systems*, pages 19–35. Springer, 2021.

20. J. Oglio, K. Hood, G. Sharma, and M. Nesterenko. Consensus on unknown torus with dense byzantine faults. *arXiv preprint arXiv:2303.12870*, 2023.
21. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
22. A. Pelc and D. Peleg. Broadcasting with locally bounded byzantine faults. *Information Processing Letters*, 93(3):109–115, 2005.
23. D. Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.
24. M. O. Rabin. Randomized byzantine generals. In *24th annual symposium on foundations of computer science (sfcs 1983)*, pages 403–409. IEEE, 1983.
25. L. Tseng and N. H. Vaidya. Fault-tolerant consensus in directed graphs. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 451–460, 2015.
26. K. Winkler, M. Schwarz, and U. Schmid. Consensus in rooted dynamic networks with short-lived stability. *Distributed Computing*, 32(5):443–458, 2019.